How to get started hacking NetBSD

Taylor R Campbell
riastradh@NetBSD.org

BSDCan 2024
Ottawa, Canada
June 1, 2024

How to get started hacking NetBSD

https://www.NetBSD.org/gallery/presentations/riastradh/bsdcan2024/getstarted.pdf

How to get started hacking NetBSD

```
cvs -d anoncvs@anoncvs.NetBSD.org:/cvsroot co -P src
```

Alternatively, with Git or Mercurial:

- `git clone https://github.com/NetBSD/src`
- `hg clone https://anonhg.NetBSD.org/src`

```
cd src
./build.sh -O ../obj -U -u -m alpha -N1 -j4 release
```

How I got started hacking NetBSD

- Lived in the Apple world through ~2008

How I got started hacking NetBSD

- Lived in the Apple world through ~2008
- Apple blocked running iTunes under gdb

How I got started hacking NetBSD

- Lived in the Apple world through ~2008
- Apple blocked running iTunes under gdb
- Offended by denial of autonomy

How I got started hacking NetBSD

- Lived in the Apple world through ~2008
- Apple blocked running iTunes under gdb
- Offended by denial of autonomy *in my own computer*

How I got started hacking NetBSD

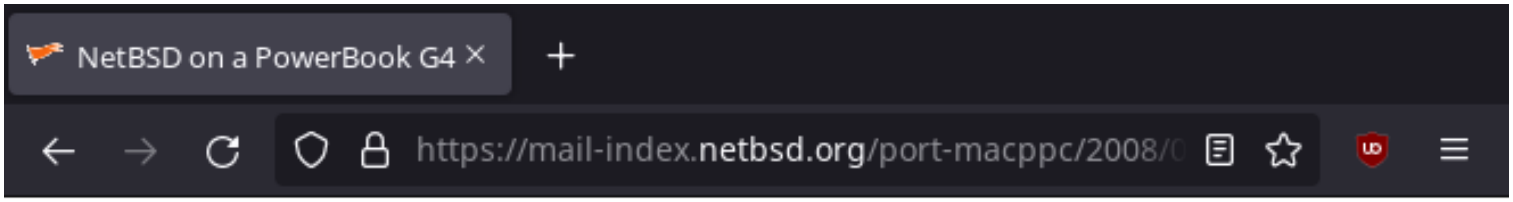
- Lived in the Apple world through ~2008
- Apple blocked running iTunes under gdb
- Offended by denial of autonomy *in my own computer*
- Shopped around for something better

How I got started hacking NetBSD

- Lived in the Apple world through ~2008
- Apple blocked running iTunes under gdb
- Offended by denial of autonomy *in my own computer*
- Shopped around for something better
- . . . something that would respect my autonomy

How I got started hacking NetBSD

- Lived in the Apple world through ~2008
- Apple blocked running iTunes under gdb
- Offended by denial of autonomy *in my own computer*
- Shopped around for something better
- . . . something that would respect my autonomy
- . . . and would run on my PowerBook G4



Port-macppc archive

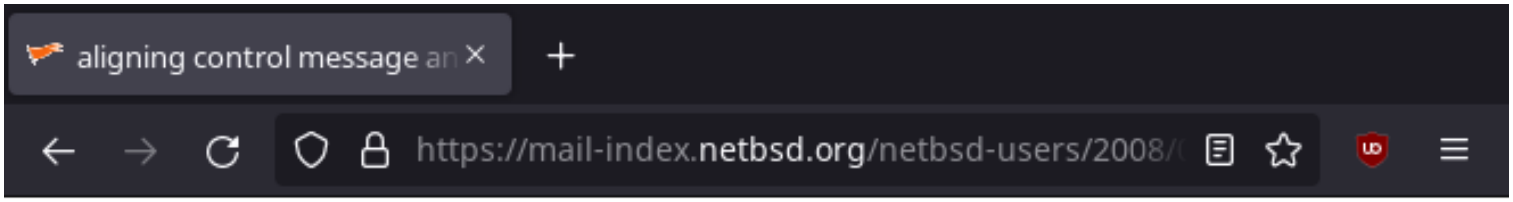
[Date Prev][Date Next][Thread Prev][Thread Next][Date Index][Thread Index][Old Index]

NetBSD on a PowerBook G4

- **To**: **port-macppc%NetBSD.org@localhost**
- **Subject**: **NetBSD on a PowerBook G4**
- **From**: **Taylor R Campbell <campbell%mumble.net@localhost>**
- Date: Thu, 5 Jun 2008 18:15:34 -0400

```
Hello!  Last night I installed NetBSD 4.0 on my PowerBook G4 1 GHz,
12-inch, and, despite the apparent complexity of the installation
instructions, once I understood how to follow only one thread through
the instructions for six different installation media and three
different major Open Firmware versions, the installation went
pleasantly smoothly.  Thanks!

I'd like to be able to experiment with this laptop as a primary work
machine.  There are two show-stopping issues with this proposition
that a preliminary Googling answered with disappointing results,
though, as did the INSTALL file for the 4.0 distribution.  This
PowerBook has an AirPort Extreme card, which is listed as unsupported;
and if I am to use this for general-purpose work, support for sleep
would be very convenient, but appears to be absent.  Is there work in
progress in it, and is this likely to change soon?  Supposing that you
were to stumble upon someone with a great deal of free time and
interest in making it change soon, what would the changes entail?
```



NetBSD-Users archive

[Date Prev][Date Next][Thread Prev][Thread Next][Date Index][Thread Index][Old Index]

aligning control message ancillary data

- **To**: **netbsd-users%netbsd.org@localhost**
- **Subject**: **aligning control message ancillary data**
- **From**: **Taylor R Campbell <campbell%mumble.net@localhost>**
- Date: Tue, 17 Jun 2008 16:41:00 -0400

Hello! Last night while debugging an issue in stunnel, I found that NetBSD (perhaps specifically NetBSD/macppc) is much more finnicky than the other operating systems I tried concerning alignment of control message ancillary data. Setting up a msghdr structure with the following idiom, in order to send a file descriptor over a Unix-domain socket using only the facilities provided in the SUSv3 and documented in the recvmsg(3) man page, causes bogus file descriptors to be received on the other end:

```
struct msghdr msg;
struct cmsghdr * cm_ptr;
struct {
  struct cmsghdr c_msghdr;
  int fd;
} control;

msg.msg_control = (void *) &control;
```

How I got started hacking NetBSD

- Read kernel code for the first time
- Just C code, guessed how syscalls are implemented
- Found bug in cmsg(3) API for fd passing
- Proposed fix, committed by christos@

How I got started hacking NetBSD

- Read kernel code for the first time
- Just C code, guessed how syscalls are implemented
- Found bug in cmsg(3) API for fd passing
- Proposed fix, committed by christos@
- . . . not quite fixed until later; fd passing is hard

How I got started hacking NetBSD device drivers

- Spent July and August porting bwi(4) from OpenBSD/DragonflyBSD
- Got it working on my PowerBook G4 Airport Extreme by September

How I got started hacking NetBSD device drivers

- Spent July and August porting bwi(4) from OpenBSD/DragonflyBSD
- Got it working on my PowerBook G4 Airport Extreme by September
- . . . didn't do a great job, not a wifi expert

How to get started hacking NetBSD

Check out source tree (a few gigabytes):

- `cvs -d anoncvs@anoncvs.NetBSD.org:/cvsroot co -P src`
- `git clone https://github.com/NetBSD/src`
- `hg clone https://anonhg.NetBSD.org/src`

How to get started hacking NetBSD

Build a release:

```
./build.sh -O ../obj -U -u -m alpha -N1 -j4 release
```

How to get started hacking NetBSD

Build a release:

```
./build.sh -O ../obj -U -u -m alpha -N1 -j4 release
```

`-O ../obj` build products go here

How to get started hacking NetBSD

Build a release:

```
./build.sh -O ../obj -U -u -m alpha -N1 -j4 release
```

`-O ../obj` build products go here

`-U` unprivileged build (traditionally privileged in /usr/src)

How to get started hacking NetBSD

Build a release:

```
./build.sh -O ../obj -U -u -m alpha -N1 -j4 release
```

`-O ../obj` build products go here

`-U` unprivileged build (traditionally privileged in /usr/src)

`-u` update build, don't clean

How to get started hacking NetBSD

Build a release:

```
./build.sh -O ../obj -U -u -m alpha -N1 -j4 release
```

`-O ../obj` build products go here

`-U` unprivileged build (traditionally privileged in /usr/src)

`-u` update build, don't clean

`-m alpha` build for DEC Alpha architecture
(use `./build.sh list-arch` to see all options)

How to get started hacking NetBSD

Build a release:

```
./build.sh -O ../obj -U -u -m alpha -N1 -j4 release
```

`-O ../obj` build products go here

`-U` unprivileged build (traditionally privileged in /usr/src)

`-u` update build, don't clean

`-m alpha` build for DEC Alpha architecture
(use `./build.sh list-arch` to see all options)

`-N1` verbosity level 1

How to get started hacking NetBSD

Build a release:

```
./build.sh -O ../obj -U -u -m alpha -N1 -j4 release
```

`-O ../obj` build products go here

`-U` unprivileged build (traditionally privileged in /usr/src)

`-u` update build, don't clean

`-m alpha` build for DEC Alpha architecture
(use `./build.sh list-arch` to see all options)

`-N1` verbosity level 1

`-j4` 4-way parallel build

How to get started hacking NetBSD

Build a release:

```
./build.sh -O ../obj -U -u -m alpha -N1 -j4 release
```

`-O ../obj`	build products go here
`-U`	unprivileged build (traditionally privileged in /usr/src)
`-u`	update build, don't clean
`-m alpha`	build for DEC Alpha architecture (use `./build.sh list-arch` to see all options)
`-N1`	verbosity level 1
`-j4`	4-way parallel build
`release`	build NetBSD release—kernel, userland, sets, images (use `./build.sh help` to see all operations)

What's in your NetBSD release build

`../obj/tooldir.NetBSD-9.3-amd64`
: cross-compiler toolchain

`../obj/releasedir/${MACHINE_ARCH}`
: release products, including distribution sets, kernels, and install/live media

`../obj/destdir.${MACHINE}`
: staged NetBSD installation

`../obj/usr.bin/find`
: build tree for find(1) from src/usr.bin/find

`../obj/sys/arch/${MACHINE}/compile/GENERIC`
: build tree for GENERIC kernel, including `netbsd` kernel image (may vary on some ports)

Other useful build.sh targets

`./build.sh help`

`./build.sh list-arch`

`./build.sh tools`
: build just cross-compiler toolchain

`./build.sh distribution`
: build just (tools and) userland but not kernels or release

`./build.sh sets`
: build just distribution sets

`./build.sh modules`
: build just kernel modules

Build a kernel

1. `./build.sh [...] tools`
2. `./build.sh [...] kernel=GENERIC`
   - Note: For 64-bit Arm and 64-bit RISC-V, use `kernel=GENERIC64`, not `kernel=GENERIC`
   - Some ports have various special-purpose kernels, not GENERIC, such as evbppc TWRP1025

Custom kernel config

- Local custom changes to GENERIC in sys/arch/.../conf/GENERIC.local (or sys/arch/.../conf/GENERIC64.local)
- Custom kernel config in sys/arch/.../conf/MYKERNEL

Example sys/arch/amd64/conf/DEBUG:

```
include "arch/amd64/conf/GENERIC"

options  DEBUG
options  KERNHIST
options  LOCKDEBUG
options  UVMHIST
```

Boot aarch64 in qemu

```
# Create a disk image
mkdir ~/vm
cd ~/obj/releasedir/evbarm-aarch64/binary/gzimg
gunzip -c <arm64.img.gz >~/vm/disk.img

# Make it a sparse 10 GB image
cd ~/vm
dd if=/dev/zero of=disk.img oseek=10g bs=1 count=1

# Make a symlink to the kernel
KERNDIR=~/obj/sys/arch/evbarm/compile/GENERIC64
ln -sfn $KERNDIR/netbsd.img .
```

Start qemu-system-aarch64

```
qemu-system-aarch64 \
    -kernel netbsd.img \
    -append "root=dk1" \
    -M virt \
    -cpu cortex-a53 \
    -smp 2 \
    -m 1g \
    -drive if=none,file=disk.img,id=disk,format=raw \
    -device virtio-blk-device,drive=disk \
    -device virtio-rng-device \
    -nic user,model=virtio-net-pci \
    -nographic
```

If `qemu -kernel` doesn't work

- `qemu -kernel` works on some ports: alpha, arm (32-bit and 64-bit aarch64), riscv, virt68k
  - Not yet on x86 (but almost ready!)
- If not:
  - Use vnd(4) and/or rump_ffs(4) to mount disk.img on the host to update the kernel
  - Serve the kernel from the host with httpd(8) and download it on the guest with ftp(1)

gdb on live kernel under qemu

- `$ qemu-system-aarch64 [...] -s [...]`
- ```
  $ gdb netbsd.gdb
  (gdb) target remote localhost:1234
  ```
  ```

gdb on live kernel under qemu

- `$ qemu-system-aarch64 [...] -s [...]`
- `$ gdb netbsd.gdb`
  `(gdb) target remote localhost:1234`
- qemu `-s` is short for `-gdb tcp:1234`

gdb on live running kernel

```
# gdb netbsd.gdb
(gdb) target kvm /dev/mem
```

gdb and crash(8) on system core dumps

```
# gdb netbsd.gdb
(gdb) target kvm /var/crash/netbsd.n.core
```

```
# crash -M /var/crash/netbsd.n \
      -N /var/crash/netbsd.n.core
crash> bt
crash> ps
```

(see ddb(4) for more commands)

```
# dmesg -M /var/crash/netbsd.n \
      -N /var/crash/netbsd.n.core
```

Verify system core dumps work!

Force the system to crash:

```
# sysctl -w debug.crashme_enable=1
# sysctl -w debug.crashme.panic=1
```

Verify system core dumps work!

Many other crashme nodes:

- simulate panic

Verify system core dumps work!

Many other crashme nodes:

- simulate panic
- enter ddb directly

Verify system core dumps work!

Many other crashme nodes:

- simulate panic
- enter ddb directly
- recursively lock mutex(9)

Verify system core dumps work!

Many other crashme nodes:

- simulate panic
- enter ddb directly
- recursively lock mutex(9)
- enter infinite loop with interrupts blocked

Verify system core dumps work!

Many other crashme nodes:

- simulate panic
- enter ddb directly
- recursively lock mutex(9)
- enter infinite loop with interrupts blocked
- launch golang, a well-known alternative test suite for NetBSD

Run ATF tests

```
# cd /usr/tests
# atf-run | atf-report
```

Run ATF tests unprivileged

```
$ cd /usr/tests
$ atf-run | atf-report
```

- Not all ATF tests can run unprivileged
- Those that can avoid changing system configuration

Run ATF tests and save output

```
# cd /usr/tests
# atf-run | tee /var/tmp/atf-run.out | \
      atf-report | tee /var/tmp/atf-report.out
```

Run ATF tests in a chroot

```
# chroot ~/obj/destdir.amd64
chroot# cd /dev && sh MAKEDEV all
chroot# mount -t ptyfs ptyfs /dev/pts
chroot# mount -t tmpfs tmpfs /tmp
chroot# cd /usr/tests
chroot# atf-run | atf-report
```

Requires kernel at least as new as the chroot userland.
Very handy for testing pullups to release branches!

Developing one program/library at a time

```
$ cd ~/src/usr.bin/find
$ $TOOLDIR/bin/nbmake-$MACHINE_ARCH -j4 dependall
$ $TOOLDIR/bin/nbmake-$MACHINE_ARCH -j4 install
```

- Test again straight from the chroot!
- (For libraries: works only for dynamic libraries; static libraries require rebuilding downstream users too)
- One makefile per program/library, usually short
- See src/share/mk/bsd.README for more information

Changing an include file

```
$ cd ~/src/include
$ edit stdio.h
$ $TOOLDIR/bin/nbmake-$MACHINE_ARCH -j4 includes
```

Then rebuild programs and libraries that `#include <stdio.h>`

Device drivers

- Autoconf drivers: represent hardware devices NetBSD can detect and handle
- devsw entries: /dev interfaces between userland and kernel

Device drivers

- Autoconf drivers: represent hardware devices NetBSD can detect and handle
- devsw entries: /dev interfaces between userland and kernel
- Sometimes correspond

Anatomy of an autoconf driver

Driver for hardware that can be detected by bus enumeration

```
struct foodev_softc {
        device_t        sc_self;
        kmutex_t        sc_lock;
        ...
};

CFATTACH_DECL2_NEW(foodev, sizeof(struct foodev_softc),
    foodev_match, foodev_attach, foodev_detach,
    NULL, NULL, NULL);
```

(Some day we'll switch to C99 designated initializers!)

Anatomy of an autoconf driver

```
static int
foo_match(device_t self, cfmatch_t match, void *aux)
{
        const struct pci_attach_args *pa = aux;
        ...
}

static int
foo_attach(device_t parent, device_t self, void *aux)
{ ... }

static int
foo_detach(device_t self, int flags)
{ ... }
```

Anatomy of an autoconf driver

`foo_match` Can this driver handle this device? If so, with what priority versus other drivers that can?

`foo_attach`
- Allocate resources for the device's driver state
- Expose the device to any other kernel interfaces
- Scan for children if any

`foo_detach` Device has been removed. Disconnect any users in other kernel interfaces and free resources.

See `https://www.NetBSD.org/gallery/presentations/riastradh/eurobsdcon2022/opendetach.pdf` for more on tricky issues with detach!

Anatomy of a /dev character special

Interface between userland and kernel identified by major and minor number stored in character special on disk

```
const struct cdevsw foo_cdevsw = {
        .d_open = foo_open,
        .d_close = foo_close,
        .d_read = foo_read,
        .d_write = foo_write,
        ...
        .d_cfdriver = &foodev_cd,
        .d_devtounit = dev_minor_unit,
        ...
};
```

(Don't forget to add to src/sys/conf/majors, or the appropriate machine-dependent majors list, and etc/MAKEDEV!)

Cloning devices

- Traditional /dev nodes correspond to a single hardware device with per-device state:
  - /dev/wd0 corresponds to first ATA hard drive
  - /dev/sd0 corresponds to first SCSI hard drive
  - /dev/ttyU0 corresponds to first USB serial port
- **Cloning devices** have per-open state:
  - /dev/audio virtual mixed-audio interface
  - /dev/dri/card0 graphics rendering interface
  - /dev/vhci USB virtual host controller interface

Cloning devices

```
const struct cdevsw foo_cdevsw = {
        .d_open = foo_open,
        .d_close = noclose,
        .d_read = noread,
        .d_write = nowrite,
        ...
};

static const struct fileops foo_fileops = {
        .fo_name = "foo",
        .fo_read = foo_read,
        .fo_write = foo_write,
        ...
};
```

Cloning devices

```
static int
foo_open(dev_t d, int flags, int fmt, struct lwp *l)
{
        struct file *fp;
        int fd;
        int error;

        error = fd_allocfile(&fp, &fd);
        if (error)
                return error;
        ...
        error = fd_clone(fp, fd, flags,
            &foo_fileops, privatedata);
        KASSERT(error == EMOVEFD);
        return error;
}
```

Access to device registers: bus_space(9)

```
bus_space_tag_t bst;
bus_space_handle_t bsh;
uint32_t ctl;
int error;

bst = args->bst;
error = bus_space_map(bst, args->addr, args->size, 0,
    &bsh);
if (error)
        goto fail;

ctl = bus_space_read_4(bst, bsh, FOO_CTL);
if (ctl & FOO_BROKEN)
        goto broken;
ctl |= FOO_ENABLED;
bus_space_write_4(bst, bsh, FOO_CTL, foo);
```

Handy macros for device register fields

```
#include <sys/cdefs.h>

#define RK_V1CRYPTO_TRNG_CTRL   0x0200 /* TRNG Control */
#define RK_V1CRYPTO_TRNG_CTRL_OSC_ENABLE __BIT(16)
#define RK_V1CRYPTO_TRNG_CTRL_CYCLES     __BITS(15,0)
...
ctrl = RK_V1CRYPTO_TRNG_CTRL_OSC_ENABLE;
ctrl |= __SHIFTIN(100, RK_V1CRYPTO_TRNG_CTRL_CYCLES);
RKC_WRITE(sc, RK_V1CRYPTO_TRNG_CTRL, ctrl);
```

Exposing memory to devices: bus_dma(9)

DMA: Direct memory access

- Allocate buffers at addresses that are safe for DMA
- Map buffers to bus addresses with IOMMU
  - E.g., map user buffer from write(2) so ethernet device can copy it out to the network
- Handle bouncing if buffers can't be mapped directly

Testing kernel components with Rump

- Rump runs unmodified kernel components and device drivers in userland processes, by setting up state just like a kernel
- NetBSD test suite uses Rump extensively to test kernel components, such as file systems
- Edit, compile, test kernel components from userland:
  - run `nbmake-$MACHINE` dependall/install from src/sys/rump (or selective subdirectories)
  - redo atf-run/report from chroot

Now get hacking NetBSD!

Questions?