How I learned to stop worrying and yank the USB

Taylor R Campbell
riastradh@NetBSD.org

EuroBSDcon 2022
Vienna, Austria
September 17, 2022

How I learned to stop worrying and yank the USB

https://www.NetBSD.org/gallery/presentations/riastradh/eurobsdcon2022/opendetach.pdf

Devices in BSD: autoconf(9) and /dev nodes

- autoconf(9) instances in kernel: pci0, pchb0, ppb1, wsdisplay0, xhci2, ...
  - Bundle of related driver state for a hardware device
  - Organized in a tree based on hardware
  - Discovered at boot by bus enumeration and on hotplug events
  - match, attach, detach
- /dev nodes (chardevs, blockdevs) for userland interface: /dev/uhid0, /dev/ttyU1, /dev/rsd3a, /dev/zero, ...
  - Software interface for userland (char) or file systems (block)
  - State may be:
    - backed by autoconf instance
    - allocated in software: 'cloning devices'
    - stateless: /dev/zero, /dev/null, /dev/mem, ...
  - Access bracketed by open and close as files
  - open, read/write/ioctl/strategy/..., close

autoconf example: ualea(4)

```
static int
ualea_match(device_t parent, cfdata_t match, void *aux)
{
        struct usbif_attach_arg *uiaa = aux;

        if (usb_lookup(ualea_devs, uiaa->uiaa_vendor, uiaa->uiaa_product))
                return UMATCH_VENDOR_PRODUCT;

        return UMATCH_NONE;
}

static void
ualea_attach(device_t parent, device_t self, void *aux)
{
        struct usbif_attach_arg *uiaa = aux;
        struct ualea_softc *sc = device_private(self);
        ...
}

static int
ualea_detach(device_t self, int flags)
{
        ...
}

CFATTACH_DECL_NEW(ualea, sizeof(struct ualea_softc),
    ualea_match, ualea_attach, ualea_detach, NULL);
```

cdevsw example: ulpt(4)

```
static int
ulptopen(dev_t dev, int flag, int mode, struct lwp *l)
{
        struct ulpt_softc *sc = device_lookup_private(&ulpt_cd, ULPTUNIT(dev));

        if (sc == NULL)
                return ENXIO;
        ...
}

static int
ulptclose(dev_t dev, int flag, int mode, struct lwp *l)
{
        ...
}

static int
ulptread(dev_t dev, struct uio *uio, int flags)
{
        struct ulpt_softc *sc = device_lookup_private(&ulpt_cd, ULPTUNIT(dev));
        ...
}

...

const struct cdevsw ulpt_cdevsw = {
        .d_open = ulptopen,
        .d_close = ulptclose,
        .d_read = ulptread,
        .d_write = ulptwrite,
        ...
};
```

```
pci0
  i915drmkms0
    intelfb0
      wsdisplay0 <- /dev/ttyE0
  xhci0
    usb0
      uhub0
        umass0
          scsibus0
            sd0 <- /dev/sd0a, /dev/sd0b, ...
        umass1
          scsibus1
            sd1 <- /dev/sd1a, /dev/sd1b, ...
    usb1
      uhub1
        uftdi0
          ucom0 <- /dev/ttyU0, /dev/dtyU0
  ...
```

device_t	*/dev node (amd64)*
uhid$N$	/dev/uhid$N$ (chr maj=66 min=N)
ucom$N$	/dev/ttyUN (chr maj=66 min=N) /dev/dtyUN (chr maj=66 min=0x80000 | N)
sd$N$	/dev/sd$N$a (blk maj=4 min=$64N$) /dev/sd$N$b (blk maj=4 min=$64N+1$) $\vdots$ /dev/rsd$N$a (chr maj=13 min=$64N$) /dev/rsd$N$b (chr maj=13 min=$64N+1$) $\vdots$
(cloning)	/dev/audio$N$ (chr maj=42 min=0x80 | N)
(stateless)	/dev/null

Easy timeline

1. `foo_attach` when device plugged in
2. `foo_open` when program opens /dev node
3. `foo_read`/`write`/`ioctl` when program does I/O on file
4. `foo_close` when program closes file
5. `foo_detach` when device unplugged after no longer in use

Easy timeline

1. attach
2. open
3. read/write/ioctl
4. close
5. detach

~~Easy~~ Naive timeline

1. attach
2. open
3. read/write/ioctl
4. close
5. detach

Complication: device yanked while open?

1. attach
2. open
3. read/write/ioctl
4. detach
5. more read/write/ioctl
6. close

Complication: no device to open?

1. open
2. attach
3. detach
4. open

Complication: no device to open?

1. open $\Longrightarrow$ must fail
2. attach
3. detach
4. open $\Longrightarrow$ must fail

Complication: device yanked in the middle of open?

1. attach
2. open called
3. detach
4. open returns

Complication: device yanked in the middle of open?

1. attach
2. open called
3. detach
4. open returns
   - success?

Complication: device yanked in the middle of open?

1. attach
2. open called
3. detach
4. open returns
   - success?
   - failure?

Complication: device yanked in the middle of open?

1. attach
2. open called
3. detach
4. open returns
   - success?
   - failure?
   - crash?

Complication: concurrent open?

1. attach
2. Thread 1
   - 2.1 open
   - 2.2 read/write/ioctl
   - 2.3 close
3. detach

Thread 2

2.1 open

Complication: concurrent open?

1. attach
2. Thread 1
   - 2.1 open
   - 2.2 read/write/ioctl
   - 2.3 close
3. detach

Thread 2

2.1 open

- succeed? (multi-open?)

Complication: concurrent open?

1. attach
2. Thread 1
   - 2.1 open
   - 2.2 read/write/ioctl
   - 2.3 close
3. detach

Thread 2

- 2.1 open
  - succeed? (multi-open?)
  - fail? (exclusive only?)

Complication: concurrent open?

1. attach
2. Thread 1
   - 2.1 open
   - 2.2 read/write/ioctl
   - 2.3 close
3. detach

Thread 2

2.1 open

- succeed? (multi-open?)
- fail? (exclusive only?)
- crash? (oops)

Complication: concurrent open and close?

If opened multiple times, `struct cdevsw::d_close` is called for *last* close only, until next open.

1. attach
2. T1: open
3. T2: open
4. T1: close
5. T2: close
6. detach

Complication: concurrent open and close?

If opened multiple times, `struct cdevsw::d_close` is called for *last* close only, until next open.

1. attach
2. T1: open $\Longrightarrow$ call `d_open`
3. T2: open
4. T1: close
5. T2: close
6. detach

Complication: concurrent open and close?

If opened multiple times, `struct cdevsw::d_close` is called for *last* close only, until next open.

1. attach
2. T1: open $\Longrightarrow$ call `d_open`
3. T2: open $\Longrightarrow$ call `d_open` again
4. T1: close
5. T2: close
6. detach

Complication: concurrent open and close?

If opened multiple times, `struct cdevsw::d_close` is called for *last* close only, until next open.

1. attach
2. T1: open $\Longrightarrow$ call `d_open`
3. T2: open $\Longrightarrow$ call `d_open` again
4. T1: close $\Longrightarrow$ no driver callback
5. T2: close
6. detach

Complication: concurrent open and close?

If opened multiple times, `struct cdevsw::d_close` is called for *last* close only, until next open.

1. attach
2. T1: open $\Longrightarrow$ call `d_open`
3. T2: open $\Longrightarrow$ call `d_open` again
4. T1: close $\Longrightarrow$ no driver callback
5. T2: close $\Longrightarrow$ call `d_close`
6. detach

Complication: open can fail

1. attach
2. open called
3. open fails
4. detach

Complication: open can fail

1. attach
2. open called $\Longrightarrow$ call `d_open`
3. open fails
4. detach

Complication: open can fail

1. attach
2. open called $\Longrightarrow$ call `d_open`
3. open fails $\Longrightarrow$ no driver callback—only on successful open
4. detach

Complication: concurrent open and close, but open fails?

1. attach
2. T1: open
3. T2: open called
4. T1: close
5. T2: open fails
6. detach

Complication: concurrent open and close, but open fails?

1. attach
2. T1: open $\Longrightarrow$ call `d_open`
3. T2: open called
4. T1: close
5. T2: open fails
6. detach

Complication: concurrent open and close, but open fails?

1. attach
2. T1: open $\Longrightarrow$ call `d_open`
3. T2: open called $\Longrightarrow$ call `d_open` again
4. T1: close
5. T2: open fails
6. detach

Complication: concurrent open and close, but open fails?

1. attach
2. T1: open $\Longrightarrow$ call `d_open`
3. T2: open called $\Longrightarrow$ call `d_open` again
4. T1: close $\Longrightarrow$ no driver callback
5. T2: open fails
6. detach

Complication: concurrent open and close, but open fails?

1. attach
2. T1: open $\Longrightarrow$ call `d_open`
3. T2: open called $\Longrightarrow$ call `d_open` again
4. T1: close $\Longrightarrow$ no driver callback
5. T2: open fails $\Longrightarrow$ call `d_close`, despite failure in *this thread*
6. detach

Detach

1. Detach triggered by yanking removable device
2. Must free resources allocated by attach

Detach

1. Detach triggered by yanking removable device
2. Must free resources allocated by attach
3. But what if device is still open?

Clearing a road for repaving

How do you clear a road for repaving?

Clearing a road for repaving

How do you clear a road for repaving?

1. Bulldoze it and lay rail for a tram line instead

Clearing a road for repaving

How do you clear a road for repaving?

1. ~~Bulldoze it and lay rail for a tram line instead~~

Clearing a road for repaving

How do you clear a road for repaving?

1. Close it off so no new cars can enter

Clearing a road for repaving

How do you clear a road for repaving?

1. Close it off so no new cars can enter
2. If existing cars are parked, leave a note they need to move

Clearing a road for repaving

How do you clear a road for repaving?

1. Close it off so no new cars can enter
2. If existing cars are parked, leave a note they need to move
3. Wait for all the cars to leave

Clearing a road for repaving

How do you clear a road for repaving?

1. Close it off so no new cars can enter
2. If existing cars are parked, leave a note they need to move
3. Wait for all the cars to leave

It is now safe to repave the road.

Clearing a road for repaving

How do you clear a road for repaving?

1. Close it off so no new cars can enter
2. If existing cars are parked, leave a note they need to move
3. Wait for all the cars to leave

It is now safe to repave the road and put in a separated bike lane.

Freeing a resource in use

How do you free a resource that may be in use?

1. Close it off so no new users can start using it
2. If existing users are sleeping indefinitely, wake them
3. Wait for all the users to finish

It is now safe to free the resource.

Detaching an open device

How do you free resources of an autoconf instance with open device nodes using it?

1. Prevent new opens
2. Interrupt pending I/O (read/write/ioctl)
3. Wait for opens and I/O to finish

It is now safe to free the resources.

Detaching an open device

How do you free resources of an autoconf instance with open device nodes using it?

1. Prevent new opens
2. Interrupt pending I/O (read/write/ioctl)
3. Wait for opens and I/O to finish

It is now safe to free the resources.
Difficult—or impossible—to get right inside a driver.

Detaching an open device

How do you free resources of an autoconf instance with open device nodes using it?

1. Prevent new opens
2. Interrupt pending I/O (read/write/ioctl)
3. Wait for opens and I/O to finish

It is now safe to free the resources.
Difficult—or impossible—to get right inside a driver.
Many drivers need this fixed. Can we make it easy to fix them all?

device_t references

```
dev_t dev;                  // maj/min num of /dev node
device_t dv;                // autoconf instance ptr
struct foo_softc *sc;       // driver private state

dv = device_lookup(&foo_cd, FOOUNIT(dev));
if (dv == NULL)
        return ENXIO;
sc = device_private(dv);
```

device_t references

```
dev_t dev;                  // maj/min num of /dev node
device_t dv;                // autoconf instance ptr
struct foo_softc *sc;       // driver private state

dv = device_lookup(&foo_cd, FOOUNIT(dev));
if (dv == NULL)
        return ENXIO;
sc = device_private(dv);

/* dv may be detached and sc freed at this point */
```

device_t references

```
dev_t dev;                  // maj/min num of /dev node
device_t dv;                // autoconf instance ptr
struct foo_softc *sc;       // driver private state

dv = device_lookup_acquire(&foo_cd, FOOUNIT(dev));
if (dv == NULL)
        return ENXIO;
sc = device_private(dv);

/* dv cannot be detached nor sc freed here */

device_release(dv);
```

device_t references and bdevsw/cdevsw d_open

```
const struct cdevsw foo_cdevsw = {
        .d_open = fooopen,
        ...
        .d_cfdriver = &foo_cd,
        .d_devtounit = dev_minor_unit,
        ...
};
```

device_t references and bdevsw/cdevsw d_open

```
static int
fooopen(dev_t dev, int flag, int mode, struct lwp *l)
{
        device_t dv = device_lookup(&foo_cd,
            dev_minor_unit(dev));
        struct foo_softc *sc;

        if (dv == NULL)
                return ENXIO;
        sc = device_private(dv);

        /* dv and sc stable until return */
        ...
}
```

device_t references and bdevsw/cdevsw d_open

- Minimal changes needed to drivers to make device_lookup safe in d_open:

  Add d_cfdriver and d_devtounit to struct cdevsw.

device_t references and bdevsw/cdevsw d_open

- Minimal changes needed to drivers to make device_lookup safe in d_open:

  Add d_cfdriver and d_devtounit to struct cdevsw.

- Note: d_devtounit must match!

device_t references and bdevsw/cdevsw d_open

- Minimal changes needed to drivers to make device_lookup safe in d_open:

  Add d_cfdriver and d_devtounit to struct cdevsw.

- Note: d_devtounit must match!
- Some prefab d_devtounit functions:
  - dev_minor_unit
  - disklabel_dev_unit
  - tty_unit

Digression: revoke(2) and tty security

- BSD-specific syscall: revoke(2)

Digression: revoke(2) and tty security

- BSD-specific syscall: revoke(2)
- On boot, getty(8) opens tty and calls login(1)

Digression: revoke(2) and tty security

- BSD-specific syscall: revoke(2)
- On boot, getty(8) opens tty and calls login(1)
- On successful authentication, login(1) chowns tty to login user

Digression: revoke(2) and tty security

- BSD-specific syscall: revoke(2)
- On boot, getty(8) opens tty and calls login(1)
- On successful authentication, login(1) chowns tty to login user
- After logout, getty(8) chowns tty back to root

Digression: revoke(2) and tty security

- BSD-specific syscall: revoke(2)
- On boot, getty(8) opens tty and calls login(1)
- On successful authentication, login(1) chowns tty to login user
- After logout, getty(8) chowns tty back to root
  $\Longrightarrow$ user can't open tty

Digression: revoke(2) and tty security

- BSD-specific syscall: revoke(2)
- On boot, getty(8) opens tty and calls login(1)
- On successful authentication, login(1) chowns tty to login user
- After logout, getty(8) chowns tty back to root
  $\Longrightarrow$ user can't open tty
- getty(8) then revokes tty

Digression: revoke(2) and tty security

- BSD-specific syscall: revoke(2)
- On boot, getty(8) opens tty and calls login(1)
- On successful authentication, login(1) chowns tty to login user
- After logout, getty(8) chowns tty back to root
  $\Longrightarrow$ user can't open tty *anew*
- getty(8) then revokes tty
  $\Longrightarrow$ user's *existing* opens of tty cease to work

Detaching an open device: revoke

- Detach function must revoke open instances before freeing
  - via `vdevgone` on the device major number and minor number range
- Forces d_close to be called

Closing an open file in use

What if read, write, or ioctl is still in progress when close happens?

Closing an open file in use

What if read, write, or ioctl is still in progress when close happens?

Choices of semantics:

- **Linux** Driver state lingers indefinitely until all pending I/O completes.
- **BSD** I/O is interrupted and fails immediately so driver state can be freed synchronously.

Closing an open file in use

What if read, write, or ioctl is still in progress when close happens?

Choices of semantics:

- **Linux** Driver state lingers indefinitely until all pending I/O completes.
- **BSD** I/O is interrupted and fails immediately so driver state can be freed synchronously.

Focus on BSD semantics here, not merits of choice.

Closing an open file in use

Driver must:

1. Prevent new I/O operations
2. Interrupt pending I/O operations
3. Wait for I/O to finish

It is now safe to free the driver state.

Closing an open file in use

NetBSD-current helps with this. Two approaches:

- Legacy drivers: d_close only.
- Newer drivers: d_cancel and d_close.

Legacy drivers: d_close only

On close or revoke, NetBSD-current will:

- prevent new I/O operations from starting (d_open, d_read, d_write, . . .)

Legacy drivers: d_close only

On close or revoke, NetBSD-current will:

- prevent new I/O operations from starting (d_open, d_read, d_write, . . .)
- call d_close, which must interrupt pending I/O and wait for it to complete.

Legacy drivers: d_close only

On close or revoke, NetBSD-current will:

- prevent new I/O operations from starting (d_open, d_read, d_write, . . .)
- call d_close, which must interrupt pending I/O and wait for it to complete.

Problem: Most drivers don't wait.

- Stop-gap: after d_close returns, NetBSD-current will wait for any concurrent d_open, d_read, d_write, d_ioctl, etc., before revoke(2) or vdevgone(9) returns.

Legacy drivers: d_close only

On close or revoke, NetBSD-current will:

- prevent new I/O operations from starting (d_open, d_read, d_write, . . .)
- call d_close, which must interrupt pending I/O and wait for it to complete.

Problem: Most drivers don't wait.

- Stop-gap: after d_close returns, NetBSD-current will wait for any concurrent d_open, d_read, d_write, d_ioctl, etc., before revoke(2) or vdevgone(9) returns.

Note: for drivers where d_open can hang indefinitely, such as ttys, d_close must be able to interrupt hanging d_open!

Newer drivers: d_cancel and d_close

On close or revoke, NetBSD-current will:

- prevent new I/O operations from starting (d_open, d_read, d_write, . . .)

Newer drivers: d_cancel and d_close

On close or revoke, NetBSD-current will:

- prevent new I/O operations from starting (d_open, d_read, d_write, . . .)
- call d_cancel, which must interrupt I/O and return promptly

Newer drivers: d_cancel and d_close

On close or revoke, NetBSD-current will:

- prevent new I/O operations from starting (d_open, d_read, d_write, . . .)
- call d_cancel, which must interrupt I/O and return promptly
- wait for any concurrent d_open, d_read, d_write, d_ioctl, etc., to return

Newer drivers: d_cancel and d_close

On close or revoke, NetBSD-current will:

- prevent new I/O operations from starting (d_open, d_read, d_write, . . .)
- call d_cancel, which must interrupt I/O and return promptly
- wait for any concurrent d_open, d_read, d_write, d_ioctl, etc., to return
- call d_close, which now has *exclusive access* to this device (chr/blk, major, minor)

Newer drivers: d_cancel and d_close

On close or revoke, NetBSD-current will:

- prevent new I/O operations from starting (d_open, d_read, d_write, . . .)
- call d_cancel, which must interrupt I/O and return promptly
- wait for any concurrent d_open, d_read, d_write, d_ioctl, etc., to return
- call d_close, which now has *exclusive access* to this device (chr/blk, major, minor)

This way, drivers don't need custom logic to wait for pending I/O to drain—generic kernel logic takes care of it.

Newer drivers: d_cancel and d_close

On close or revoke, NetBSD-current will:

- prevent new I/O operations from starting (d_open, d_read, d_write, . . .)
- call d_cancel, which must interrupt I/O and return promptly
- wait for any concurrent d_open, d_read, d_write, d_ioctl, etc., to return
- call d_close, which now has *exclusive access* to this device (chr/blk, major, minor)

This way, drivers don't need custom logic to wait for pending I/O to drain—generic kernel logic takes care of it.

Note: for drivers where d_open can hang indefinitely, such as ttys, d_cancel must be able to interrupt hanging d_open!
New `ttycancel` function can be used for d_cancel in most or all tty drivers.

```
static int
uhidread(dev_t dev, struct uio *uio, int flag)
{
        struct uhid_softc *sc =
            device_lookup_private(&uhid_cd, UHIDUNIT(dev));
        ...
        mutex_enter(&sc->sc_lock);
        while (sc->sc_q.c_cc == 0) {
                ...
                if (sc->sc_closing) {
                        mutex_exit(&sc->sc_lock);
                        return EIO;
                }
                error = cv_wait_sig(&sc->sc_cv,
                    &sc->sc_lock);
                if (error)
                        break;
        }
        ...
}
```

```
static int
uhidcancel(dev_t dev, int flag, int mode, struct lwp *l)
{
        struct uhid_softc *sc =
            device_lookup_private(&uhid_cd, UHIDUNIT(dev));

        if (sc == NULL)
                return 0;

        /* Interrupt pending I/O, make it fail promptly. */
        mutex_enter(&sc->sc_lock);
        sc->sc_closing = true;
        cv_broadcast(&sc->sc_cv);
        mutex_exit(&sc->sc_lock);

        uhidev_stop(sc->sc_hdev);

        return 0;
}
```

```
static int
uhid_detach(device_t self, int flags)
{
        struct uhid_softc *sc = device_private(self);
        int maj, mn;

        /* locate the major number */
        maj = cdevsw_lookup_major(&uhid_cdevsw);

        /* Forcibly close any open instances. */
        mn = device_unit(self);
        vdevgone(maj, mn, mn, VCHR);

        /* Safe to free resources now! */
        ...
}
```

Interrupted open must restart

If d_open sleeps, and d_cancel or d_close wakes it

Interrupted open must restart

If d_open sleeps, and d_cancel or d_close wakes it (e.g., in a tty driver),

Interrupted open must restart

If d_open sleeps, and d_cancel or d_close wakes it (e.g., in a tty driver), after wakeup, permissions checked before d_open may have changed,

Interrupted open must restart

If d_open sleeps, and d_cancel or d_close wakes it (e.g., in a tty driver), after wakeup, permissions checked before d_open may have changed, so d_open *must* return `ERESTART` to restart the system call and redo the permissions checks.

New driver contract: summary

Set d_cfdriver and d_devtounit to match device_lookup use in d_open; in exchange:

- detach prevents new d_open from starting
- device_lookup result in d_open is stable

Set d_cancel to interrupt pending I/O (including open) and return promptly; in exchange:

- d_close has exclusive access to (chr/blk, maj, min) triple among concurrent devsw functions
- No further I/O (including d_open) possible until d_close returns

New driver contract: summary

Set d_cfdriver and d_devtounit to match device_lookup use in d_open; in exchange:

- detach prevents new d_open from starting
- device_lookup result in d_open is stable

Set d_cancel to interrupt pending I/O (including open) and return promptly; in exchange:

- d_close has exclusive access to (chr/blk, maj, min) triple among concurrent devsw functions
- No further I/O (including d_open) possible until d_close returns

(Lots of detailed edge cases handled behind the scenes in spec_vnops.c—very hairy!)

Usage model

1. attach
2. while attached:
   (a) d_open on first open
      (i) I/O: (d_open | d_read | d_write | d_ioctl | . . .)*
      (ii) d_cancel—then NetBSD waits for existing I/O to finish
   (b) d_close on last close
3. `vdevgone` returns in detach; no more I/O possible

(for drivers with d_cancel)

Questions?