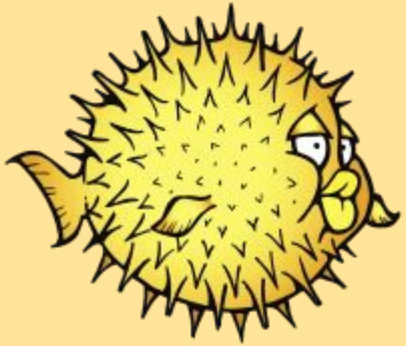

Devices, Drivers and PCI

COMP3301 - 5 Week Applied Class

OpenBSD

COMP3301 - 2025

The University of Queensland
Australia

Devices and Drivers

- A **device** is any **external hardware** that is attached to your computer.
  - Hard Disks, Graphics Cards, Mice, Keyboards, Monitors
  - Network Devices, Encryption Devices,
- Devices participate in the autoconf(9) system
  - a framework which enables devices on a system and decides what driver to use for a particular device
- Supported devices are **architecture specific**
- There are two types of devices
  - Block devices (bdev)
    - You write to them in blocks / “random access”
    - (usul. File systems, formats, hard disks swap devices)
  - Character devices (cdev)
    - You write to them in characters
    - Pretty much everything else
- Each cdev has a unique magic number among cvdevs and bdevs among bdevs

Device Special Files

- Device special files (a.k.a. device nodes) in the /dev/ directory
- Provide a user-space interface to kernel device drivers
- Have an associated major or minor number that connects to the driver

Example:

- /dev/ttyACM0 (cdev)
  - USB serial device “0” (ACM = Abstract Control Model)
  - Example: Arduino, USB modem
- /dev/sda (bdev)
  - SCSI disk “a” (Small Computer System Interface)

- devices usually create these on attaching
- mknod(8) can be used to make device special files manually

OpenBSD Basic Driver Anatomy

- const struct cfattach
- struct cfdriver
- int match(struct device *parent, void *match, void *aux)
    - Asks the driver “Can you drive this device?”
- void attach(struct device *parent, struct *device self, void *aux))
    - What do I need to set up for a device driver when a device is attached
- int detach(struct device *parent, void *match, int flags)
    - What do I need to clean up for a device driver when the device is detached
- Software context (struct softc)
    - Struct that contains context information for a particular instance of a device.
    - You decide what goes in here – anything you want to be kept by the driver

OpenBSD cdev entry points - Basics!

- int open(dev_t dev, int oflags, int devtype, struct proc *p)
  a. How do I open a session to my device?
- int close(dev_t dev, int iflags, int devtype, struct proc *p)
  a. How do I clean up a session to my device?
- int read(dev_t dev, struct uio *uio, int ioflag)
  a. How do I define “reading” from my device
- int write(dev_t dev, struct uio *uio, int ioflag)
  a. How do I define “writing to my device”
- int ioctl(dev_t dev, u_long *cmd, caddr_t data, int fflags, struct proc *p)
  a. “Input/Ouput Control”
- Not all syscalls are used for every device
  - E.g. for some devices read/write might not make sense. You may purely interface with a device via ioctls
  - Open/Close might not make sense for a device that doesn’t need per-connection states or isolation

OpenBSD cdev entry points (Continued)

- int kqueue(dev_t dev, struct knote *kn)
  - Scalable event notification interface
  - Can be used to allow non-blocking behaviour for long requests
- tty(struct tty *tp, int rw)
  - Returns the tty struct associated with the device
    (for pseudo-terminals, serial ports, console devices, etc.)
- mmap(dev_t dev, off_t offset, int flags)
  - Defined how or if a user process can memory-map your device into their address space.
    - Might be used by processes like x11 etc.
- We will go into kqueue in more detail in a future contact but tty and mmap won't be discussed in this course.

OpenBSD bdev entry points

- int open(dev_t dev, int oflags, int devtype, struct proc *p)
  - How do I open a session to my device?
- int close(dev_t dev, int iflags, int devtype, struct proc *p)
  - How do I clean up a session to my device?
- void *strategy(struct buf *bp)
  - How do I access a block of my device for I/O
- int ioctl(dev_t dev, u_long *cmd, caddr_t data, int fflags, struct proc *p)
  - “Input/Ouput Control”
- int dump(dev_t dev, u_long *cmd, caddr_t data, int fflags, struct proc *p)
  - Special: writes device contents for kernel crash dumps.

open() and close()

- int open(dev_t dev, int oflags, int devtype, struct proc *p)
  a. How do I open a session to my device?
- int close(dev_t dev, int iflags, int devtype, struct proc *p)

`dev_t dev`	The device being accessed (from /dev/) with associated Major and Minor Numbers
`int oflags /` `int iflags`	Flags specifying access mode (read, write, non-blocking, etc.)
`int devtype`	Device type (`cdev` or `bdev`)
`struct proc *p`	Pointer to the calling process

read() and write()

- int read(dev_t dev, struct uio *uio, int ioflag)
  a. How do I define "reading" from my device
- int write(dev_t dev, struct uio *uio, int ioflag)
  a. How do I define "writing to my device"

`dev_t dev`	The device being accessed (from /dev/) with associated Major and Minor Numbers
`struct uio *uio`	Pointer to an array of io requests.
`int ioflags`	Flags specifying access mode, e.g., IO_UNIT, IO_NDELAY

Struct UIO

```c
struct uio {
    struct iovec *uio_iov;    /* Pointer to array of I/O vectors */
    int          uio_iovcnt; /* Number of elements in uio_iov */
    off_t        uio_offset; /* Offset in the device or file */
    ssize_t      uio_resid;  /* Remaining bytes to transfer */
    enum uio_rw  uio_rw;     /* UIO_READ or UIO_WRITE */
    struct proc *uio_procp;  /* Pointer to process performing the I/O */
};
```

ioctl()

- int ioctl(dev_t dev, u_long *cmd, caddr_t data, int fflags, struct proc *p)

`dev_t dev`	The device being accessed (from /dev/) with associated Major and Minor Numbers
`u_long cmd`	Command code — identifies the operation requested by the user program. Each driver can define its own set of commands.
`caddr_t data`	Pointer to a data structure in user space containing arguments for the command. Driver reads/writes this to get/set device state.
`int fflags`	Flags passed from the open file descriptor (e.g., read/write mode)
`struct proc *p`	Pointer to the calling process.

ioctl() - commands

- Each ioctl() has a unique number for it's device
- Up to 13 bits
- Cmd numbers encode
  - Length of params
  - Direction of params
  - Subsystem of device
  - Group of device

```
31        30 29 28 27 ... 16 15 ... 8 7 ... 0

+------------+------+-------+----------+

| Direction  | Size | Group | Number   |

+------------+------+-------+----------+

| Read/Out| len | '5' | cmd num   |

+------------+------+-------+----------+
```

```c
#if !defined(_SYS_P5D_H)
#define _SYS_P5D_H

#include <sys/ioctl.h>
#include <sys/ioccom.h>
#include <sys/types.h>

struct p5d_status_params {
    uint     psp_is_num_waiting;
};

#define     P5D_IOC_STATUS     _IOR('5', 1, struct p5d_status_params)

#endif /* _SYS_P5D_H */
```

Peripheral Controller Interface (PCI)

- A bus device
  - Transfers data in paralell
  - High speed access to all devices on the bus (For ye olden days)
  - 133MB/s to 533MB
- PCI Express (PCIe)
  - Even faster!
  - up to 242 GB/s
- In Comparison
  - AXI -> 1GB/s
  - USB FS -> 48MB/s
  - USB HS -> 480MB/s
  - I2c -> 400KB/s
  - SPI -> 4MB/s
- Registers in the form of “Bars”
- Can have capabilities like MSIx or DMA (more on this next time)
- pcidump -v

Bars and Bus Space

- The main “registers” for pci devices are found in their “bars”
- These could be settings, or small fields for entry
- They could also be triggers (e.g. DBELL regs)

Offset	Size	Name	Description
0x00	64 bits	A	The first number to be added (R/W)
0x08	64 bits	B	The second number to be added (R/W)
0x10	64 bits	SUM	The sum of A and B (read-only) : $SUM = A + B$

Memory Mapping and Barriers

- You can directly map memory for bars to an address in your OS
- pci_mapreg_map
- For linear mapping you can organise this with a struct so you only need to map once
- Most modern systems have “caching”
  - IO operations may not be resolved immediately to improve performance
  - When external sources modify these, or try to read them they may not be up to date
- “Barriers” can be used to force IO operations to occur immediately
  - A write barrier will force the CPU to write to the register instead of leaving it in it’s cache
  - A read barrier will force a CPU to discard it’s cached value and read the memory again

Practicals for this Week and Next Week

- Prac 4 - Basic PCI Device Driver (Content from Today)
  - Writing a basic PCI driver for the emulated p4d to add two numbers
  - Redefining add2 syscall in the pci driver to interface with userland
  - End result functionally the same as prac 2
- Prac 5 - Writing a Pseudo Device (Content from Today)
  - Define a “pseduo device” in the kernel as a cdev
    - gives you an understanding of how devices are defined in the kernel
    - You don’t usually have to go through all these steps if a type of device is already defined for it (e.g. pci, usb)
  - End result functionally the same as prac 3
- Prac 6 - More Advanced PCI Device Driver (Next week’s content)
  - Writing a PCI device driver to interface with a more advanced device
  - Using DMA to write to ring buffers
  - Sending requests to a device
    - Blocking – waiting on the device for the result
    - Non-blocking - making the request then polling the device for completions
  - Catching MSIx generated interrupts for completions and handling them
  - Using kqueuefilter() to poll for completions (handling non-blocking requests)

Coming Weeks - Device Events and DMA

- PCIe - Events and Interrupts
- DMA - Direct Memory Access
- MSIx -Message Signaled Interrupts
- kqueue() - scalable event notification interface
- Assignment 2 - Write a device driver for an emulated PCIe device