

Lecture 16:

Device Drivers

CS343 – Operating Systems
Branden Ghena – Fall 2025

Some slides borrowed from:

Stephen Tarzia (Northwestern), Jaswinder Pal Singh (Princeton), and UC Berkeley CS162

Administrivia

- Filesystem Lab
 - Strongly recommend you get started on this *before* Thanksgiving
 - Especially given the number of slip days many groups spent on Paging Lab
- Due Thursday of last week of classes (December 4)
 - 2.5 weeks from today, but that includes Thanksgiving week
 - That due date is extremely unlikely to change
 - No room to push it later
 - I never extend deadlines

Today's Goals

- Explore how software for device I/O is architected.
- Discuss OS considerations at multiple software layers.
- Investigate example device drivers
 - One in Nautilus
 - One in Linux and Tock

Outline

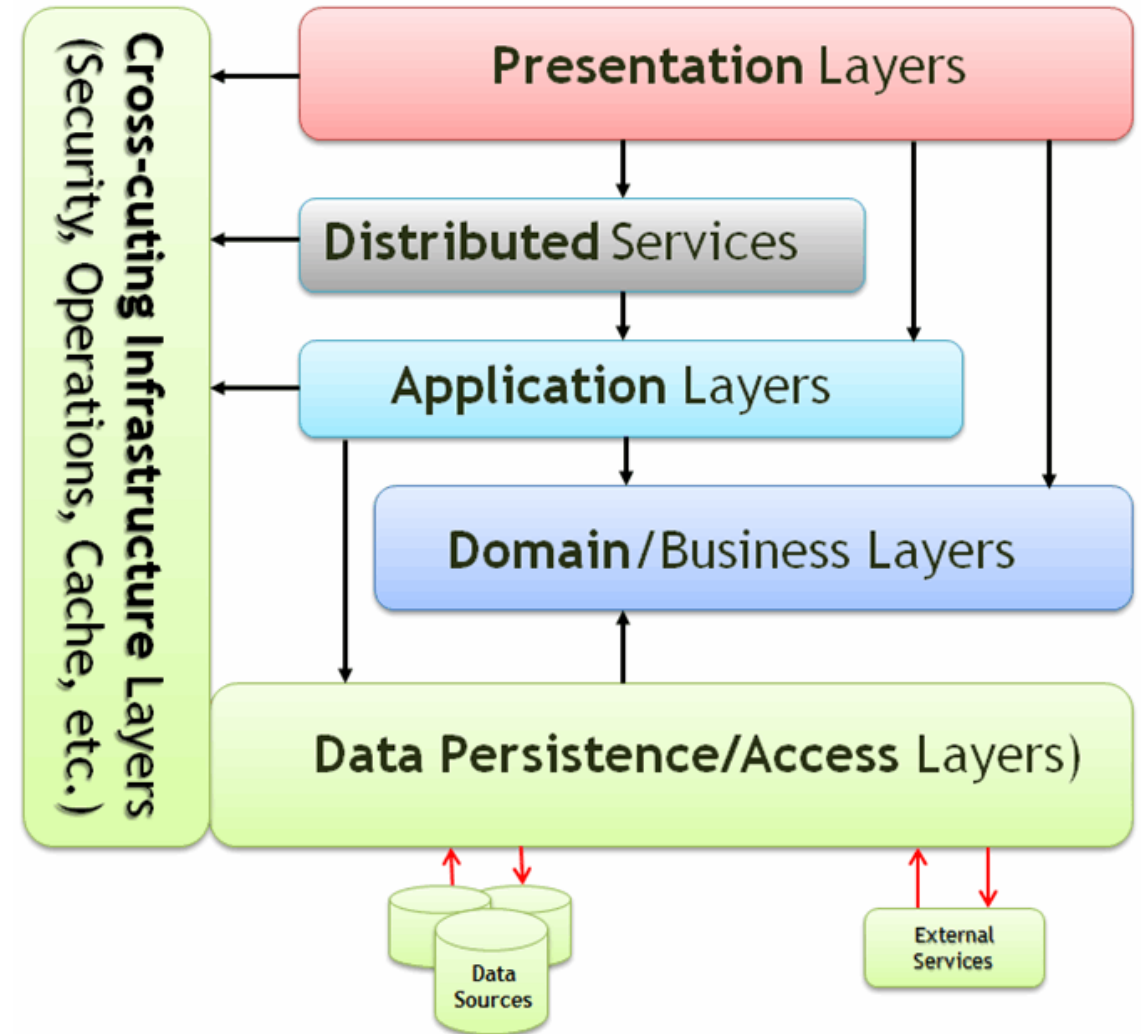
- **Abstractions**
- Device I/O layers
 - Application Layer
 - Kernel I/O Subsystem
 - Device Driver
 - Interrupt Handler
- Example Driver: Nautilus Character Device
- Example Driver: Temperature Sensor

Writing software to manage devices

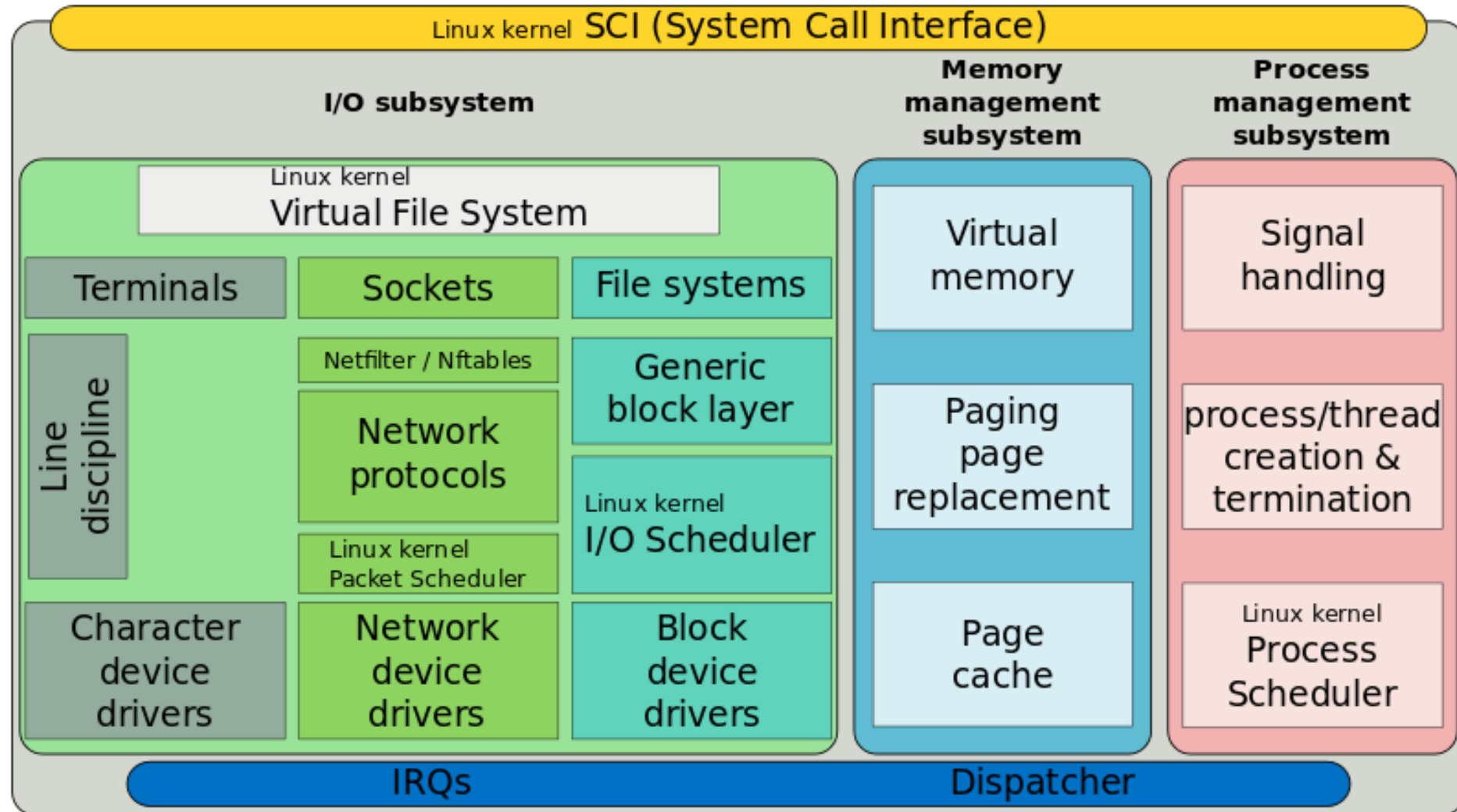
- Kernel software for managing a device is a *device driver*
 - 70% of Linux code is device drivers
 - 15.3 Million lines of source code
- Big challenge for device drivers
 - How do we enable interactions with so many varied devices?
 - Need abstractions to allow software to interact with them easily
 - Need mechanisms to reuse a lot of code for commonalities

General software abstractions

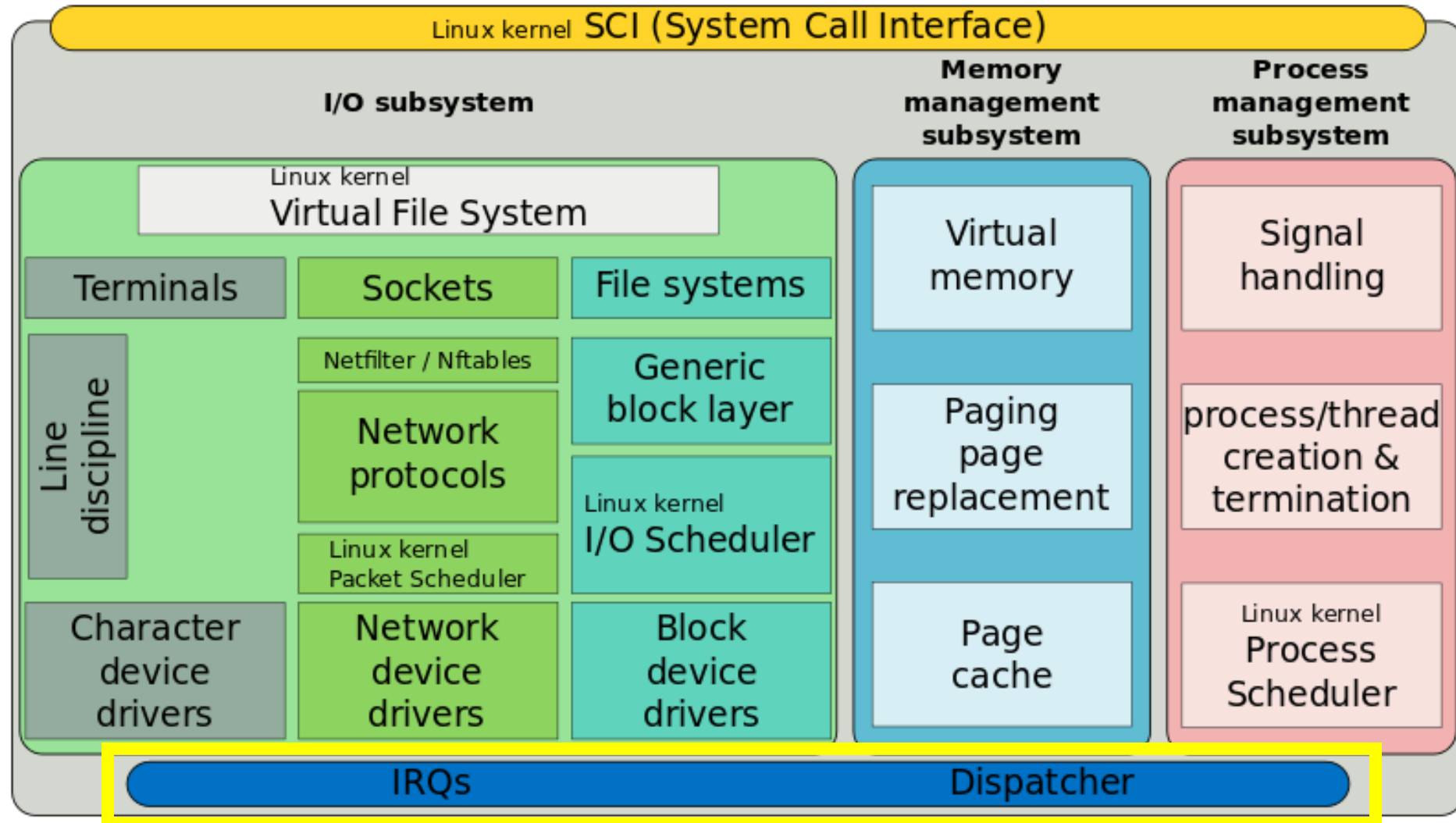
- When building large software projects, we like to define layers of code
 - Makes it clear what is handled where
 - Enables swapping out implementations when desired



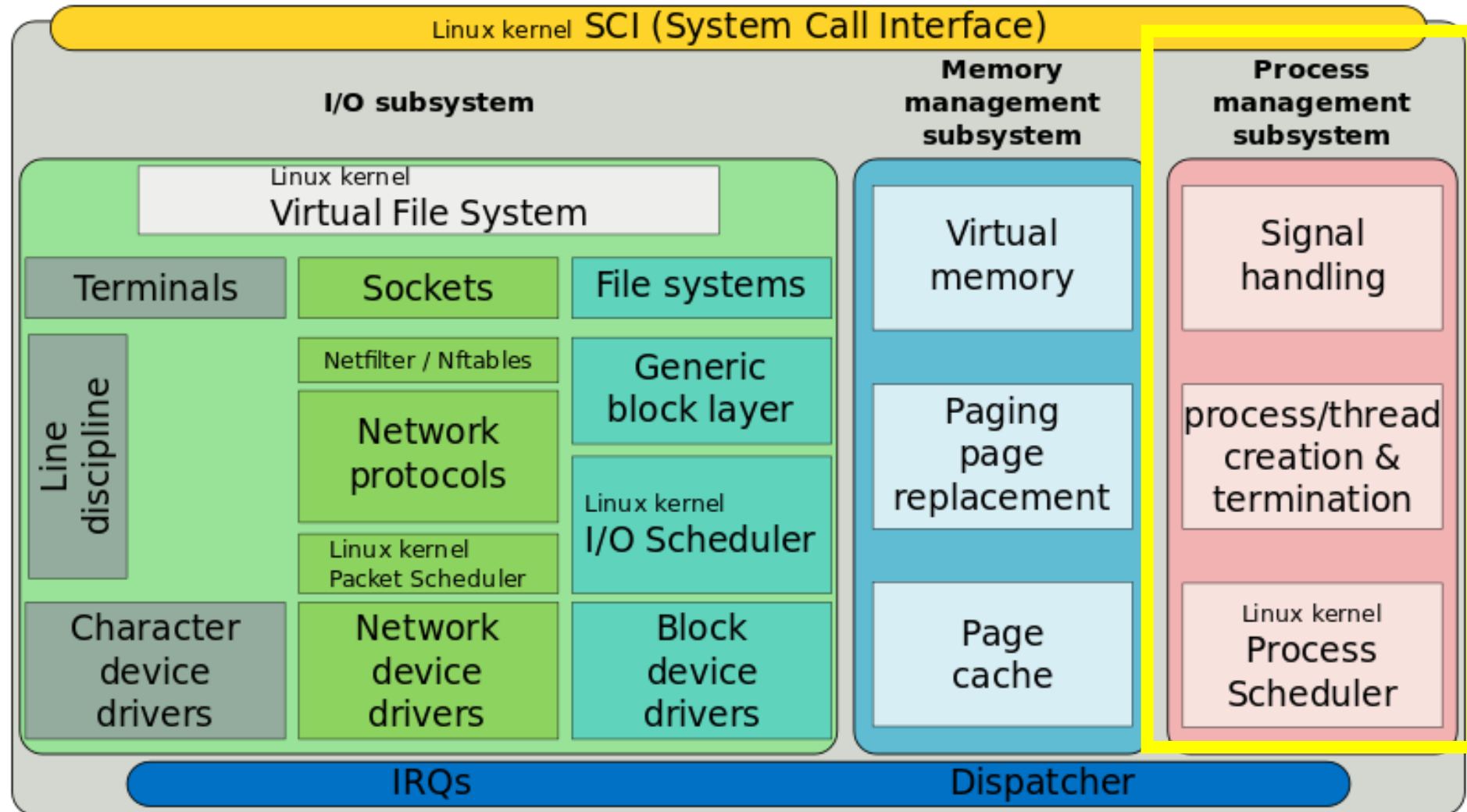
Linux Kernel Layering



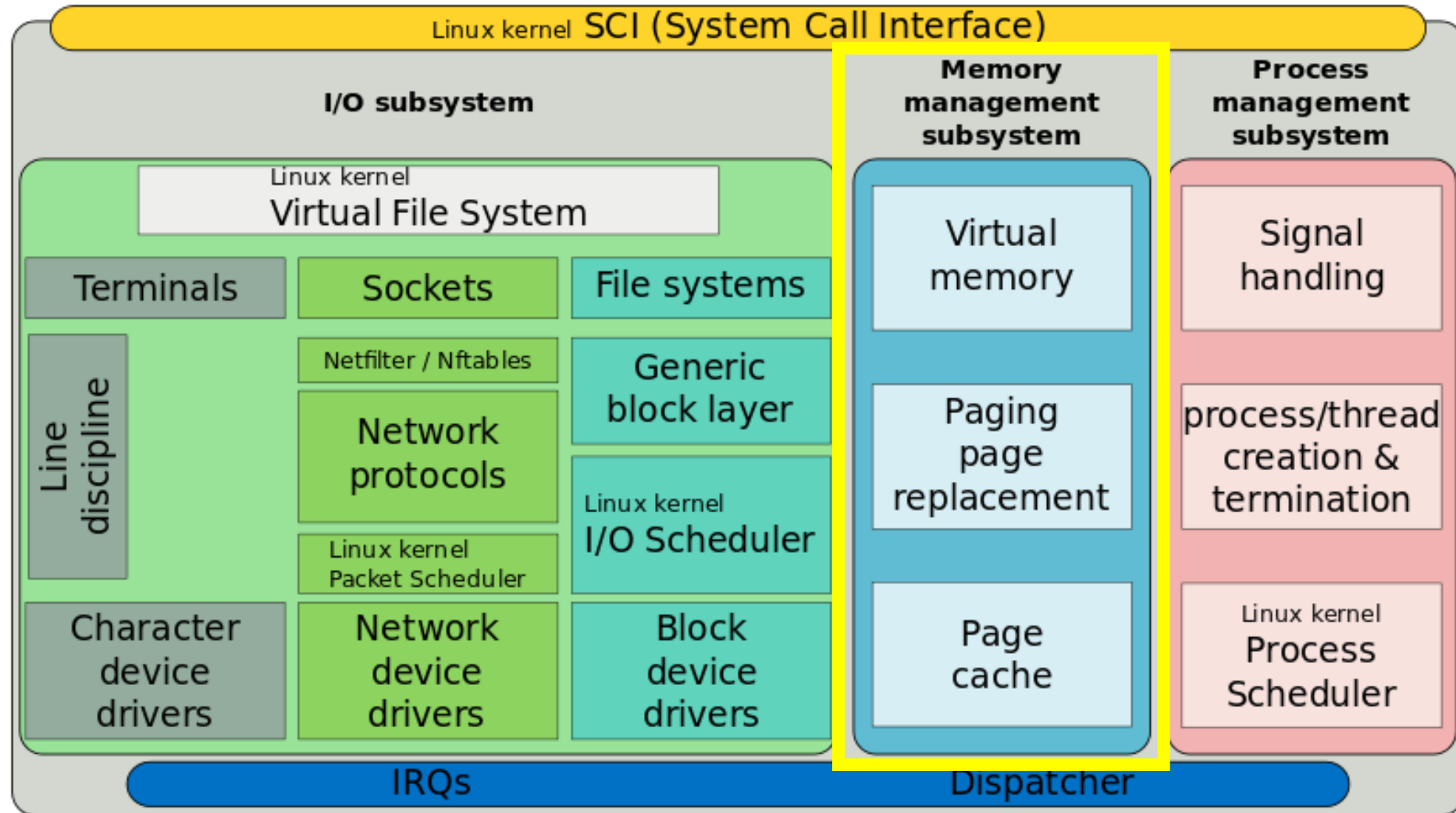
Linux Kernel Layering



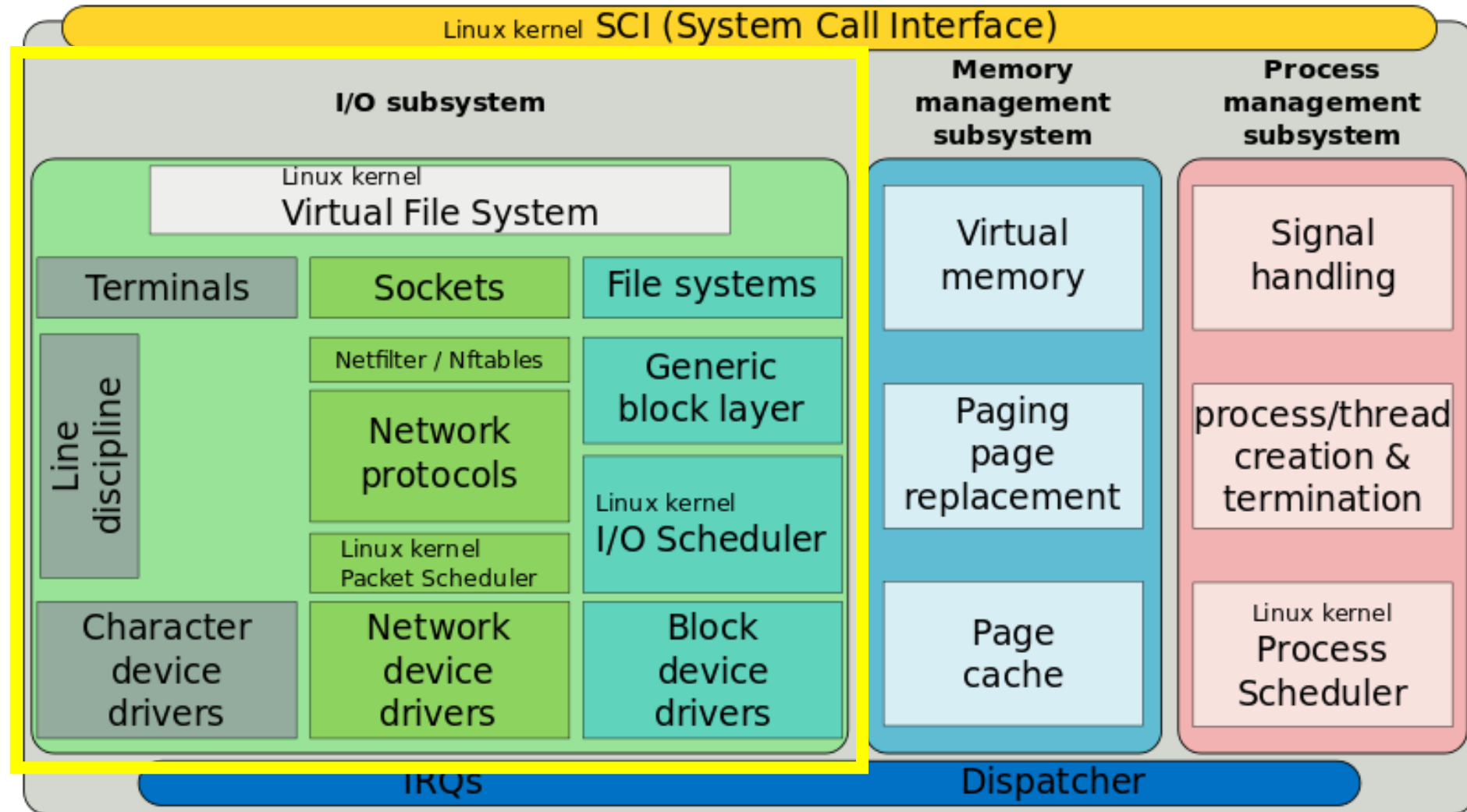
Linux Kernel Layering



Linux Kernel Layering

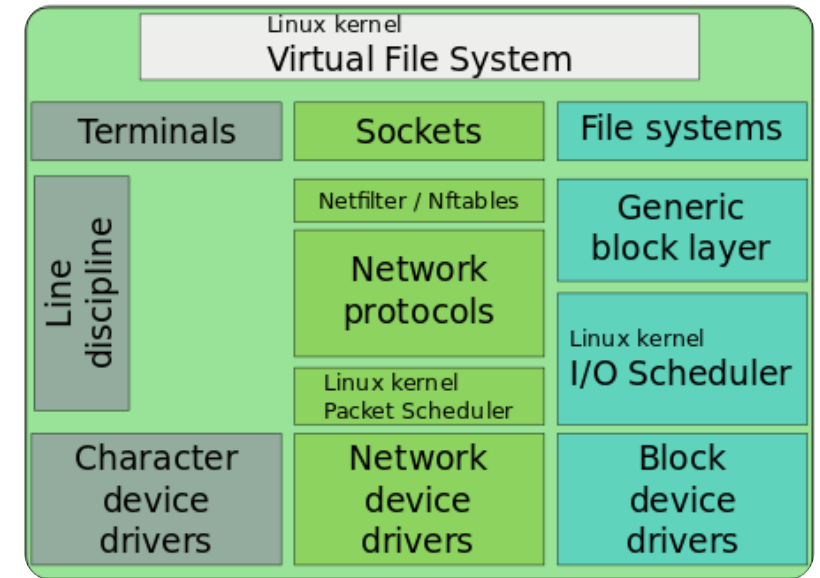


Linux Kernel Layering



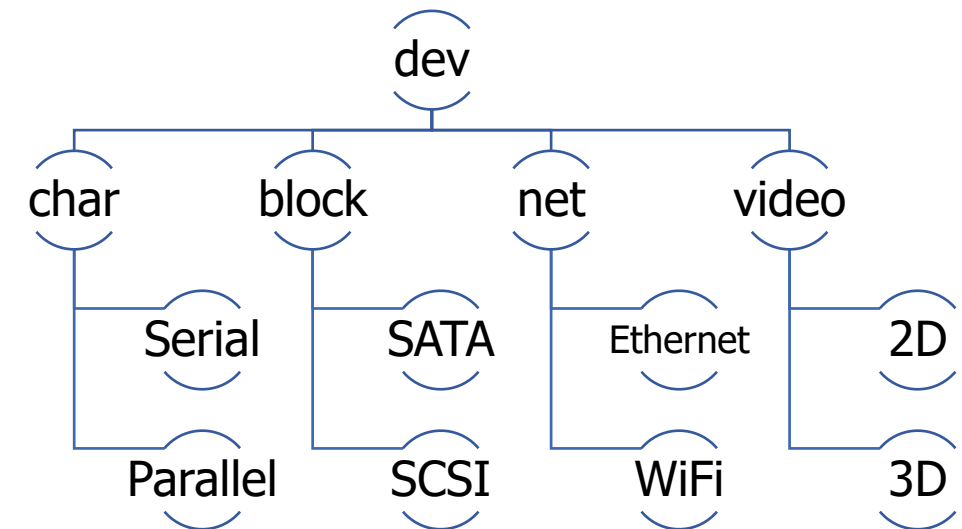
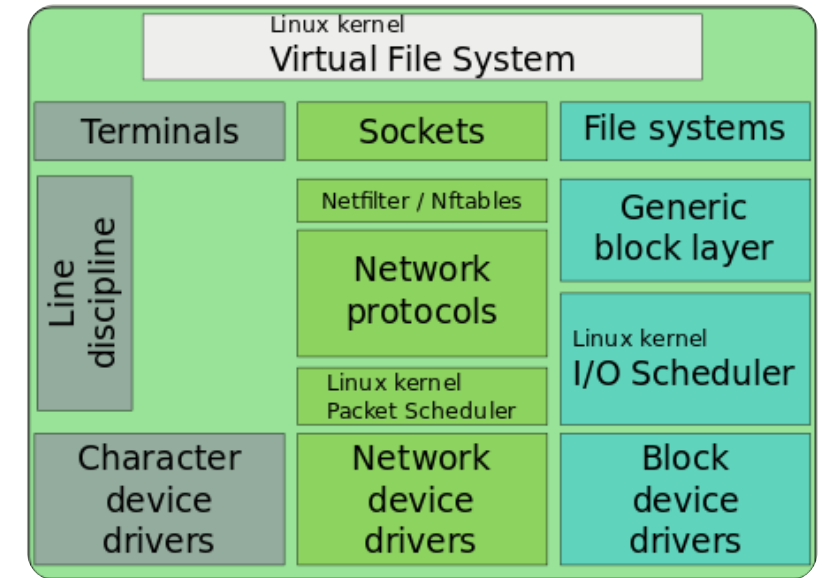
Abstraction: everything is a file!

- Hardware: treat devices like memory
 - They can be read and written at addresses
- Software: treat devices like files
 - They can be read and written
 - They may be created or destroyed (plugged/unplugged)
 - They can be created in hierarchies. Example:
 - SATA devices
 - SSD
 - USB devices
 - Webcam
 - Microphone



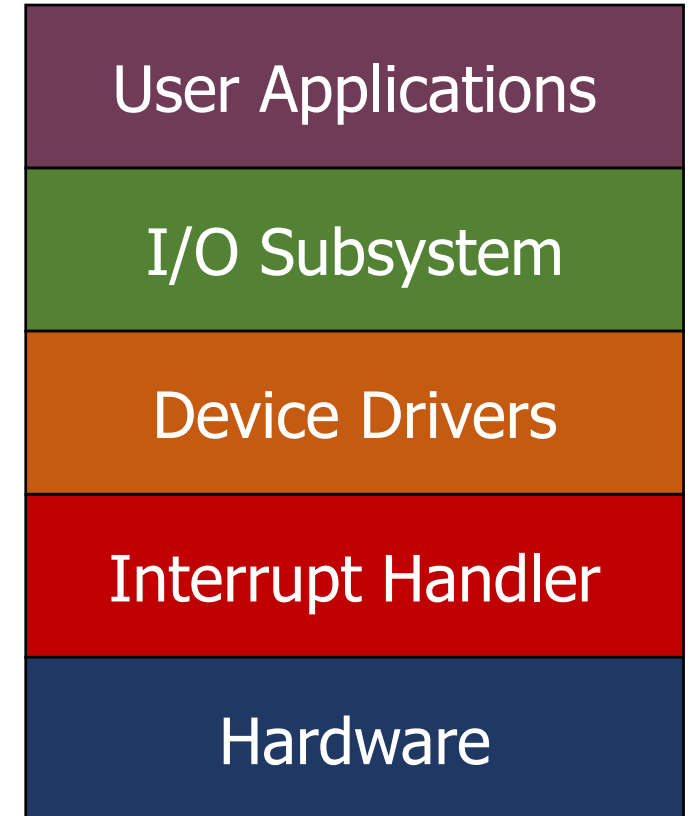
Linux device classes

- Character devices
 - Accessed as a stream of bytes (like a file)
 - Example: Webcam, Keyboard, Headphones
 - We will focus on these
- Block devices
 - Accessed in blocks of data (like a disk)
 - Can hold entire filesystems
 - Example: Disks, Flash drives
- Network interfaces
 - See CS340 (Computer Networking)
 - Accessed through transfer of data packets



System layers when interacting with devices

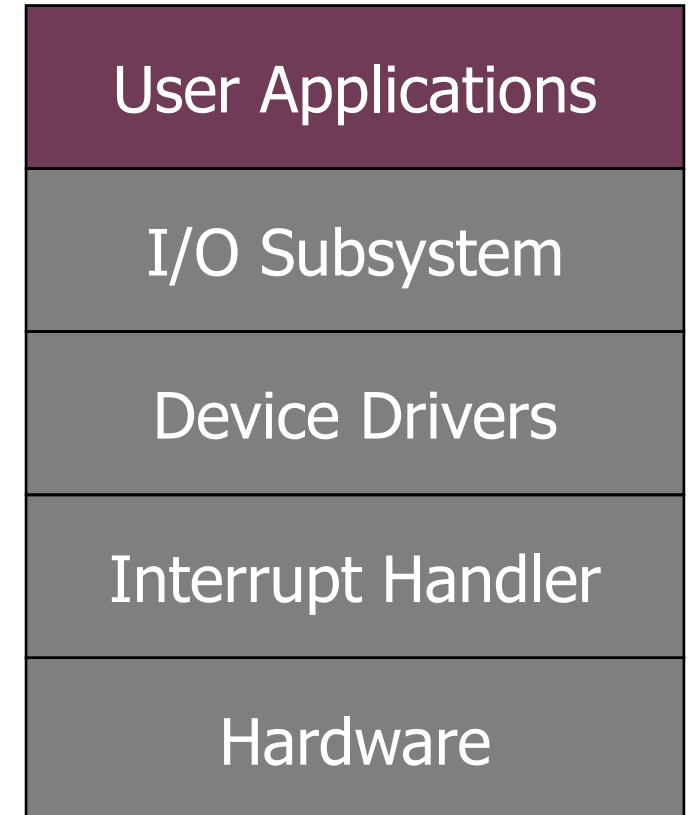
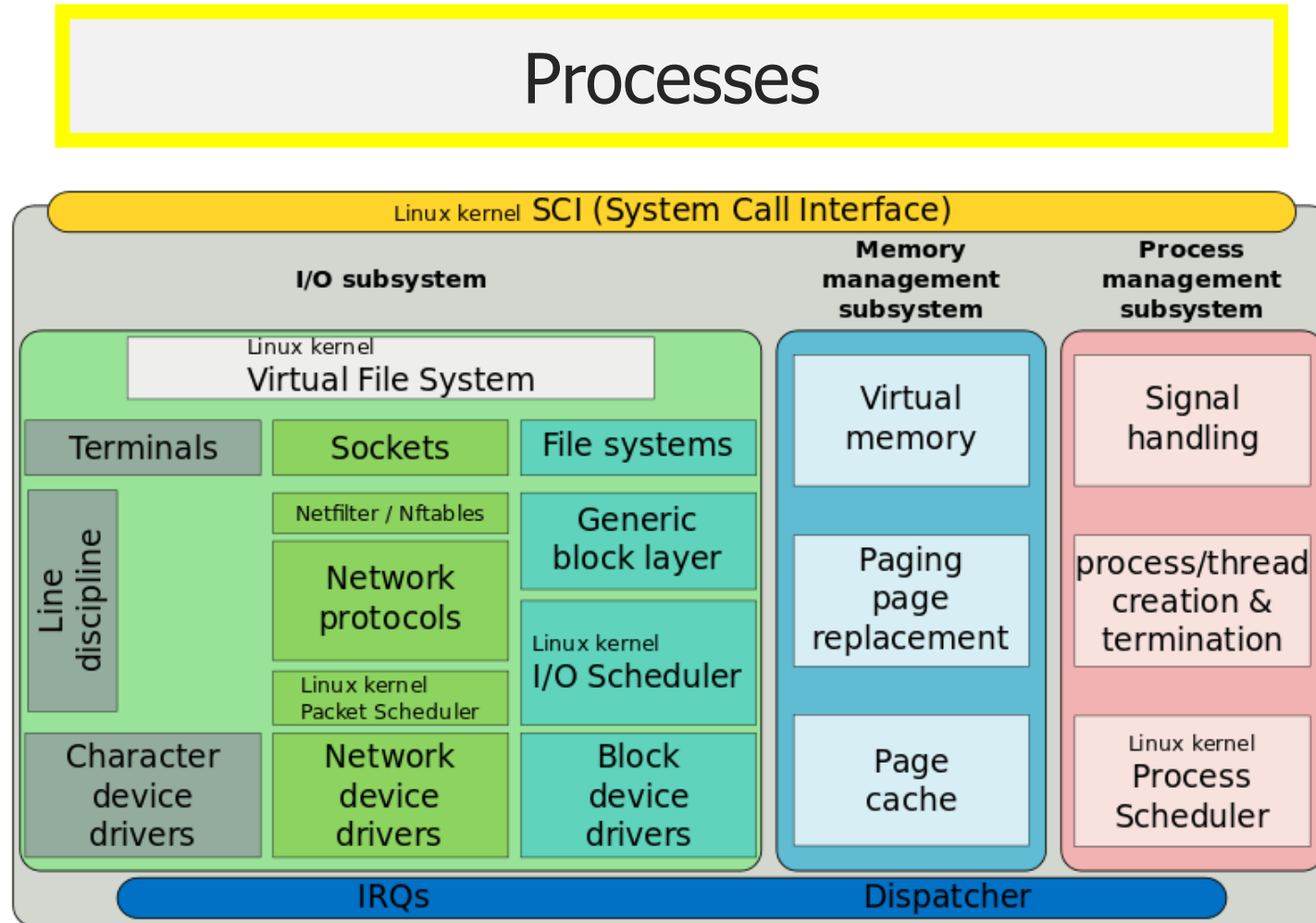
- User applications
 - Do useful things
- I/O subsystem
 - Receive syscalls, route to device drivers
- Device drivers
 - Translate application requests into device interactions
- Interrupt Handler
 - Receive events from hardware
- Hardware
 - Do useful things



Outline

- Abstractions
- **Device I/O layers**
 - **Application Layer**
 - Kernel I/O Subsystem
 - Device Driver
 - Interrupt Handler
- Example Driver: Nautilus Character Device
- Example Driver: Temperature Sensor

Where we are at in the system



Communication with devices

- Interactions occur through system calls
 - Open/Close
 - Read/Write
 - Seek, Flush
 - Ioctl
 - And various others

Accessing devices

- Open/Close
 - Inform device that something is using it (or not)
 - Argument is path to device (like path to file)
 - Get a file descriptor that the other operations act on
- “/dev” directory is populated with devices

```
[brghena@ubuntu code_examples] $ ls /dev/
agpgart  dri  lightnvm  mcelog  rtc0  tty0  tty22  tty36  tty5  tty63  ttyS18  ttyS31  vcs3  vcsu4
autofs  dvd  log  mem  sda  tty1  tty23  tty37  tty50  tty7  ttyS19  ttyS4  vcs4  vcsu5
block  ecryptfs  loop0  midi  sda1  tty10  tty24  tty38  tty51  tty8  ttyS2  ttyS5  vcs5  vcsu6
bsg  fb0  loop1  mqueue  sda2  tty11  tty25  tty39  tty52  tty9  ttyS20  ttyS6  vcs6  vfio
btrfs-control  fd  loop10  net  sda5  tty12  tty26  tty4  tty53  ttyprintk  ttyS21  ttyS7  vcsa  vga_arbiter
bus  full  loop2  null  sg0  tty13  tty27  tty40  tty54  ttyS0  ttyS22  ttyS8  vcsa1  vhci
cdrom  fuse  loop3  nvram  sg1  tty14  tty28  tty41  tty55  ttyS1  ttyS23  ttyS9  vcsa2  vhost-net
cdrw  hidraw0  loop4  port  shm  tty15  tty29  tty42  tty56  ttyS10  ttyS24  udmabuf  vcsa3  vhost-vsock
char  hpet  loop5  ppp  snapshot  tty16  tty3  tty43  tty57  ttyS11  ttyS25  uhid  vcsa4  vmci
console  hugepages  loop6  psaux  snd  tty17  tty30  tty44  tty58  ttyS12  ttyS26  uinput  vcsa5  vsock
core  hwrng  loop7  ptmx  sr0  tty18  tty31  tty45  tty59  ttyS13  ttyS27  urandom  vcsa6  zero
cpu_dma_latency  initctl  loop8  pts  stderr  tty19  tty32  tty46  tty6  ttyS14  ttyS28  userio  vcsu  zfs
cuse  input  loop9  random  stdin  tty2  tty33  tty47  tty60  ttyS15  ttyS29  vcs  vcsu1
disk  kmsg  loop-control  rfkill  stdout  tty20  tty34  tty48  tty61  ttyS16  ttyS3  vcs1  vcsu2
dmide  kvm  mapper  rtc  tty  tty21  tty35  tty49  tty62  ttyS17  ttyS30  vcs2  vcsu3
```

Interacting with devices

- Same read/write commands you've likely seen before
 - These are actually syscalls!
- Read
 - `ssize_t read(int fd, void *buf, size_t count);`
- Write
 - `ssize_t write(int fd, const void *buf, size_t count);`

Arbitrary device interactions

- ioctl – I/O Control
 - `int ioctl(int fd, unsigned long request, ...);`
- Request number followed by an arbitrary list of arguments
 - “request” may be broken in fields: command, size, direction, etc.
- Catch-all for device operations that don't fit into file I/O model
 - Combine with “magic numbers” to form some special action
 - Reset device, Start action, Change setting, etc.
 - Read the device documentation to find these

ioctl example – webcam capabilities

- Magic number to request webcam capabilities: `0x80685600`
 - Returns a struct with various data fields
 - Driver name, Device name, Bus location
 - Capabilities flags (various video modes, async operation support)
 - Documentation: <https://www.kernel.org/doc/html/v4.9/media/uapi/v4l/vidioc-querycap.html>

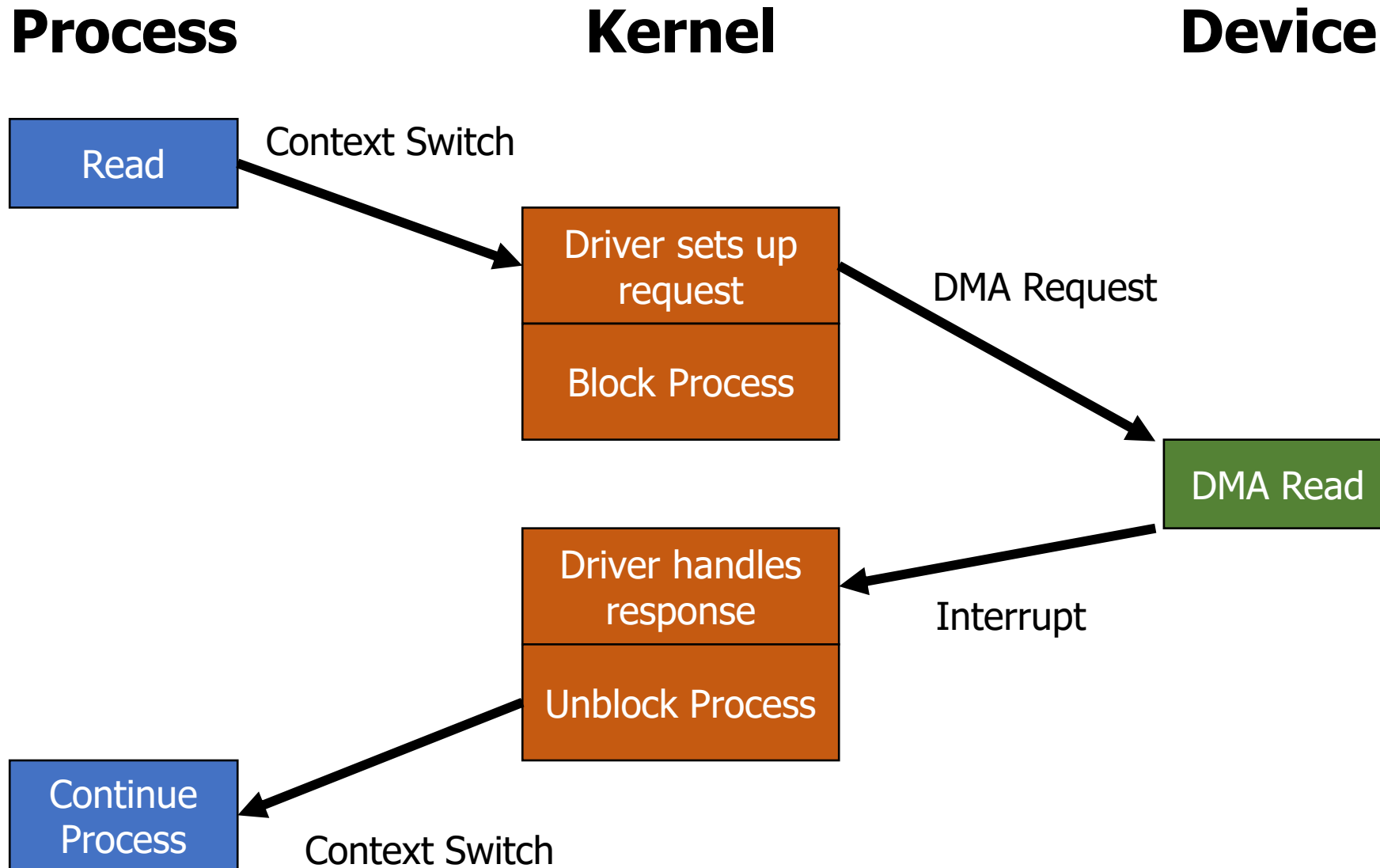
```
./ioctl /dev/video0 0x80685600 | hexdump -C
Decoded values: ioctl=0x80685600, direction=R, arg size=104 bytes, device number=0x56 ('V')
Used values: ioctl=0x80685600, direction=R, arg size=104 bytes, device number=0x56 ('V')
Returned 0
00000000  75 76 63 76 69 64 65 6f  00 00 00 00 00 00 00 00  | uvcvideo.....|
00000010  48 44 20 50 72 6f 20 57  65 62 63 61 6d 20 43 39  | HD Pro Webcam C9|
00000020  32 30 00 00 00 00 00 00  00 00 00 00 00 00 00 00  | 20.....|
00000030  75 73 62 2d 30 30 30 30  3a 30 33 3a 30 30 2e 30  | usb-0000:03:00.0|
00000040  2d 32 00 00 00 00 00 00  00 00 00 00 00 00 00 00  | -2.....|
00000050  0c 08 06 00 01 00 a0 84  01 00 20 04 00 00 00 00  | .....|
00000060  00 00 00 00 00 00 00 00  | .....|
00000068
```

Command line tool for sending ioctl commands: <https://github.com/jerome-pouiller/ioctl>

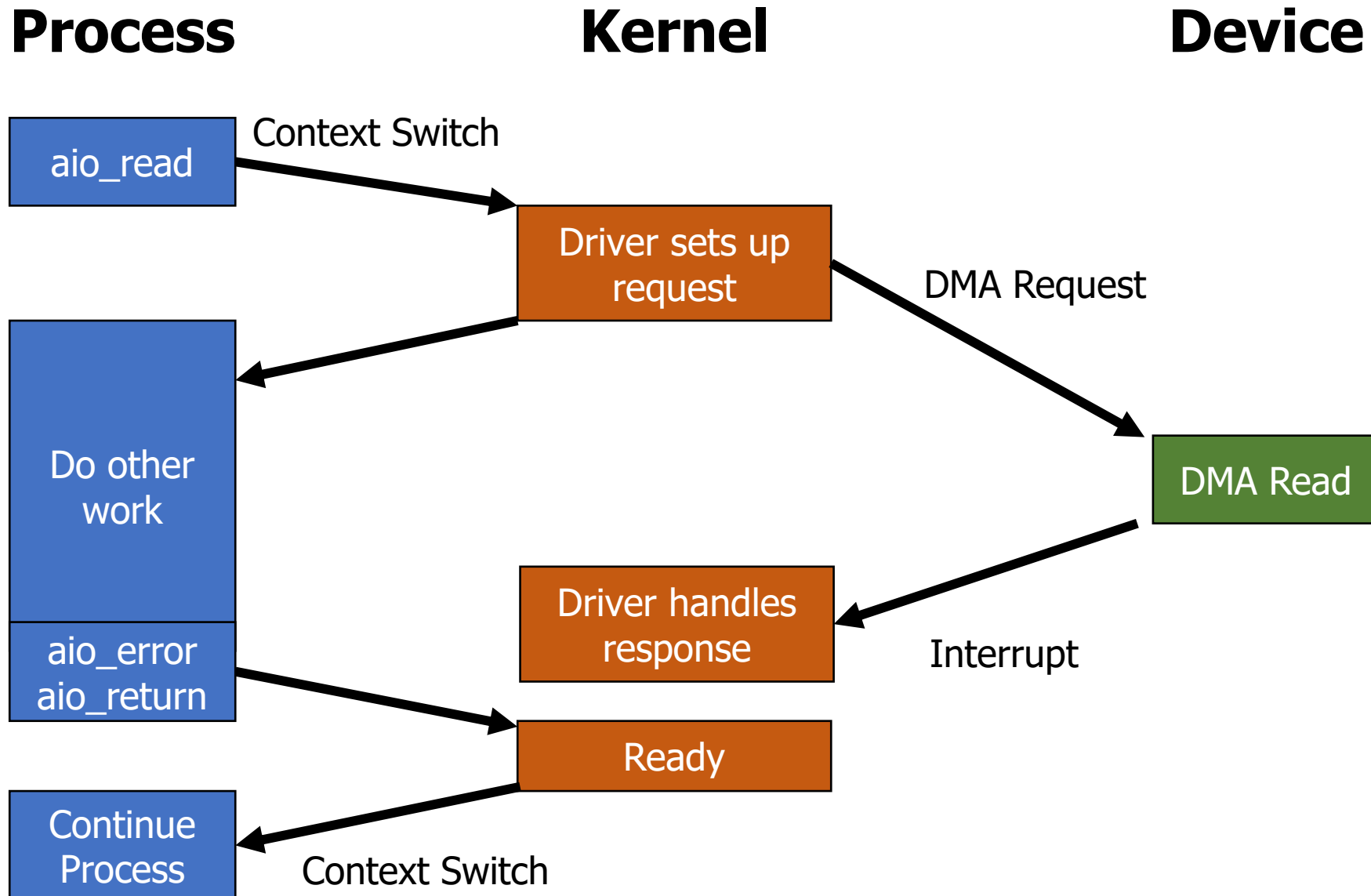
Asynchronous I/O operations

- Previous examples were all synchronous I/O calls
 - Read/Write will block process until complete
 - Easy to use, but not always most efficient method
- Asynchronous I/O calls also exist
 - POSIX AIO library
 - aio_read/aio_write – enqueue read/write request
 - aio_error – check status of an I/O request
 - aio_return – get result of a completed I/O request

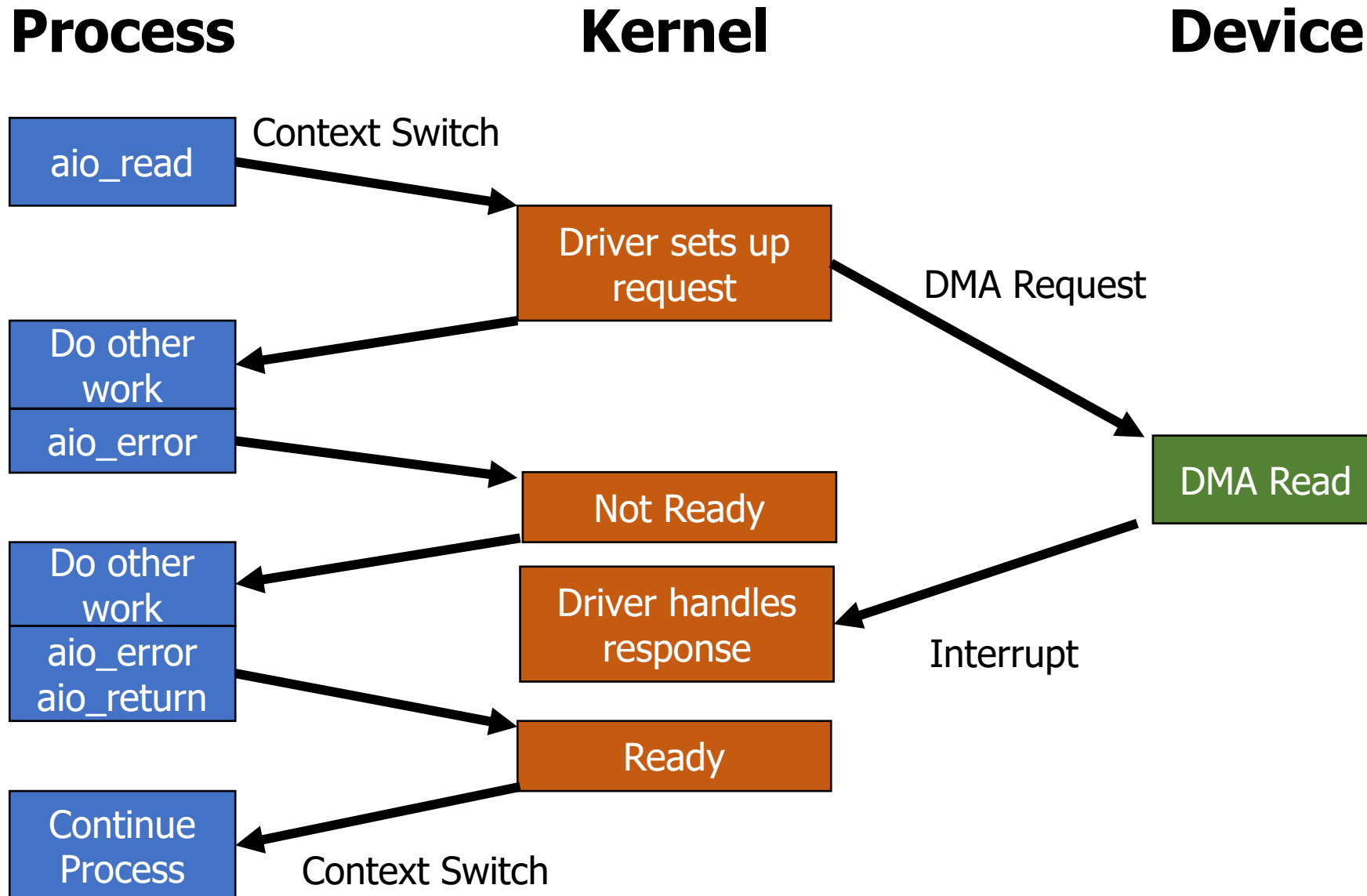
Synchronous blocking read example



Asynchronous read example



Asynchronous read example with early request



Break + Open Question

- Could you re-create the asynchronous I/O interface using threads?

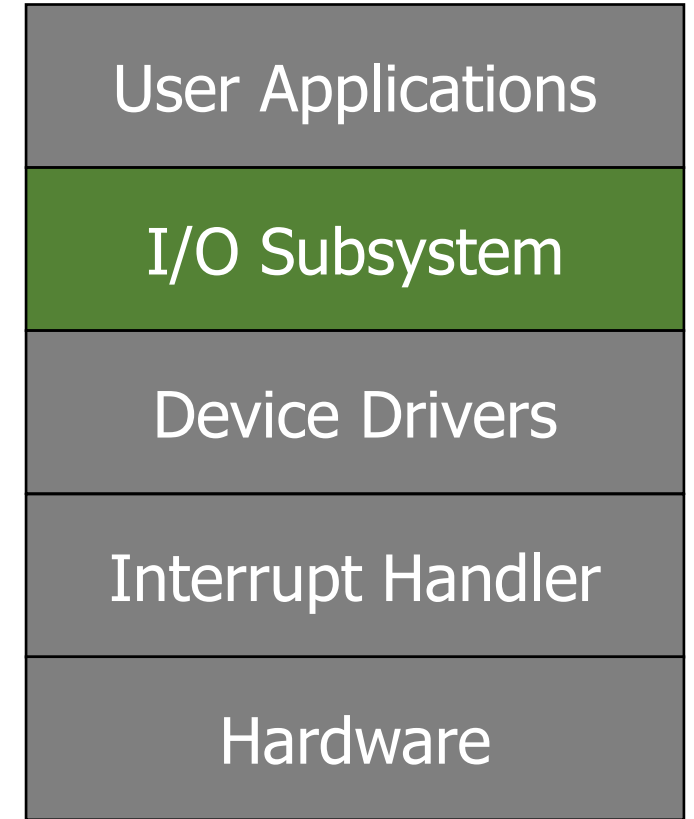
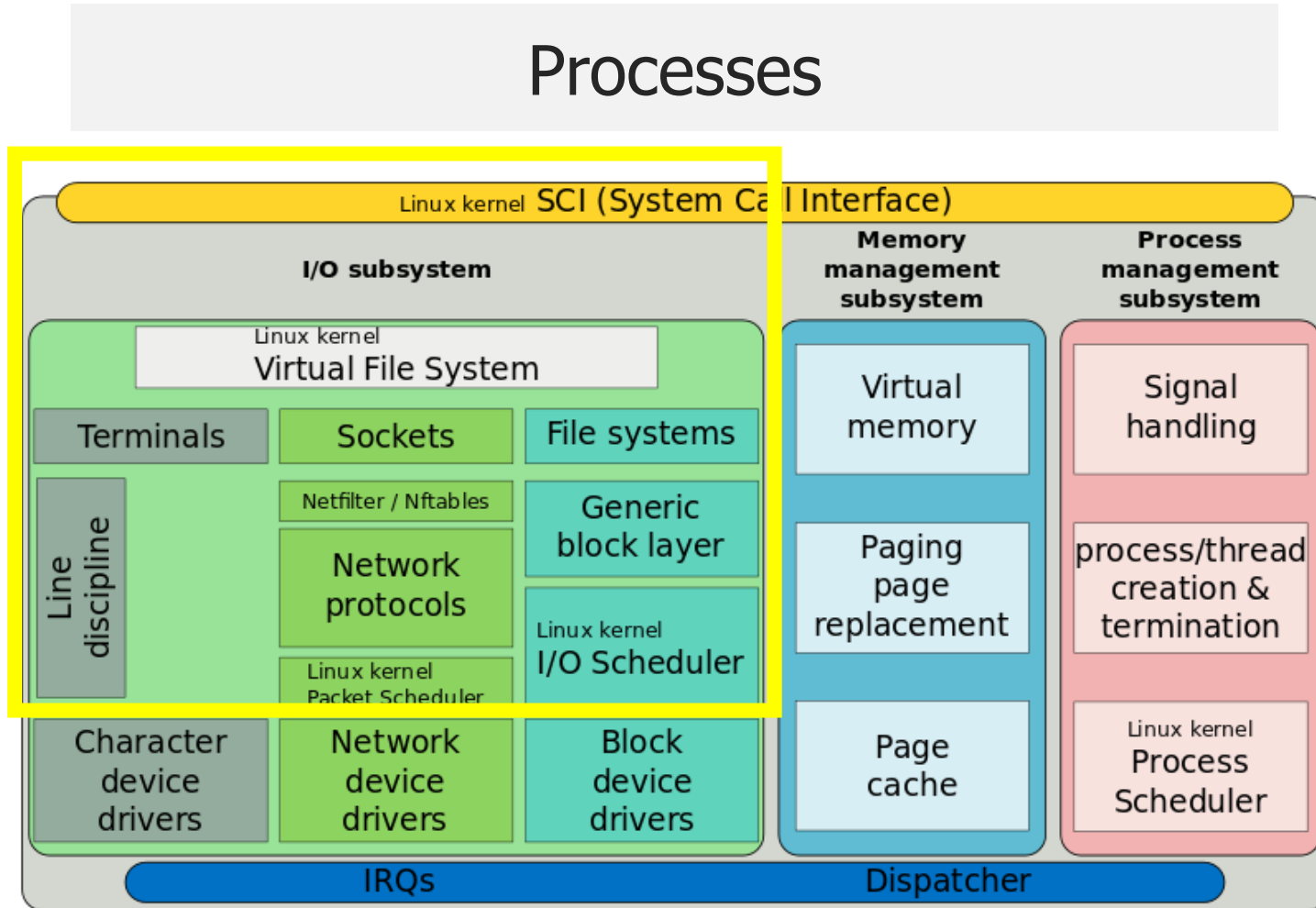
Break + Open Question

- Could you re-create the asynchronous I/O interface using threads?
 - `aio_read` creates a new thread, which does the actual blocking read
 - Thread will essentially block immediately
 - `aio_error` / `aio_return` get data from that worker thread
 - Synchronized with locks
 - Thread exits after `aio_return` occurs
- This is basically the underlying implementation for glibc POSIX AIO

Outline

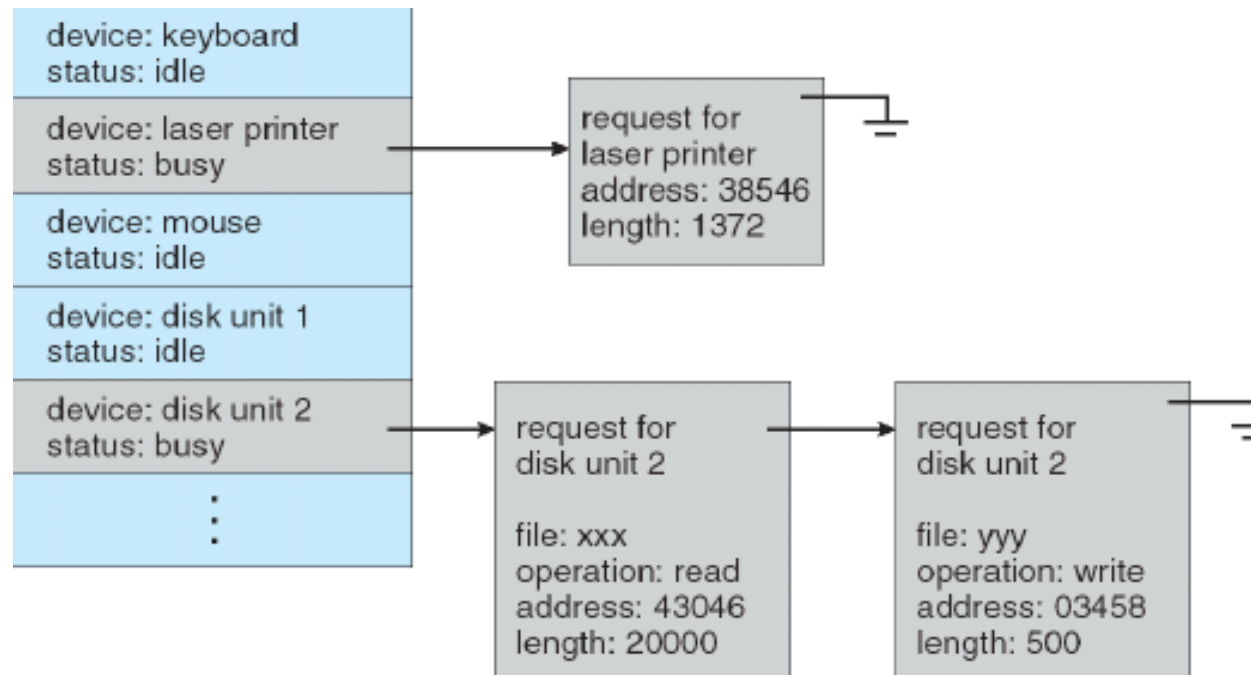
- Abstractions
- **Device I/O layers**
 - Application Layer
 - **Kernel I/O Subsystem**
 - Device Driver
 - Interrupt Handler
- Example Driver: Nautilus Character Device
- Example Driver: Temperature Sensor

Where we are at in the system



Kernel I/O subsystem

- The OS kernel does various things for devices that are not specific to the individual device
 - Manages permissions
 - Routes call to appropriate driver
 - Schedules requests to drivers



Kernel needs to handle process memory

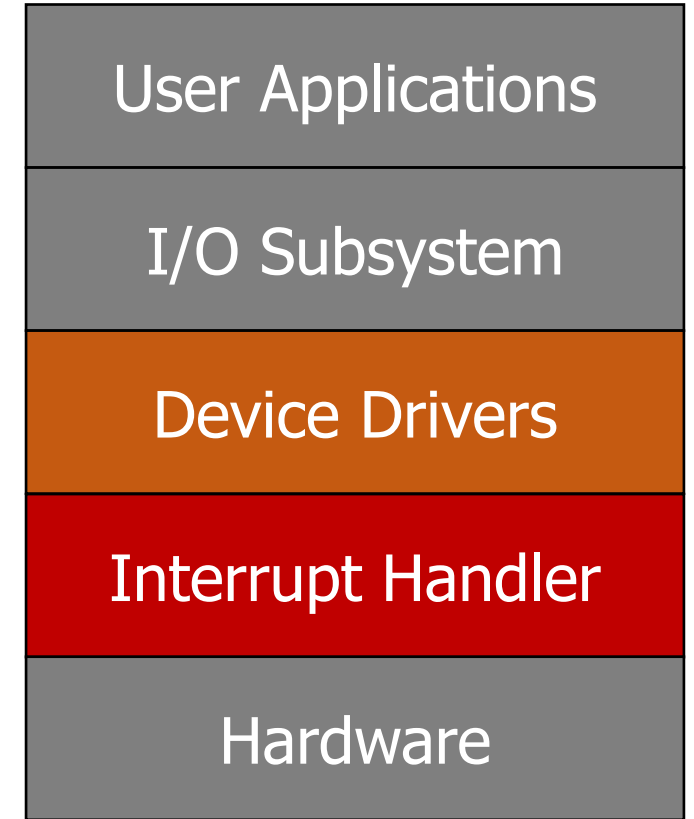
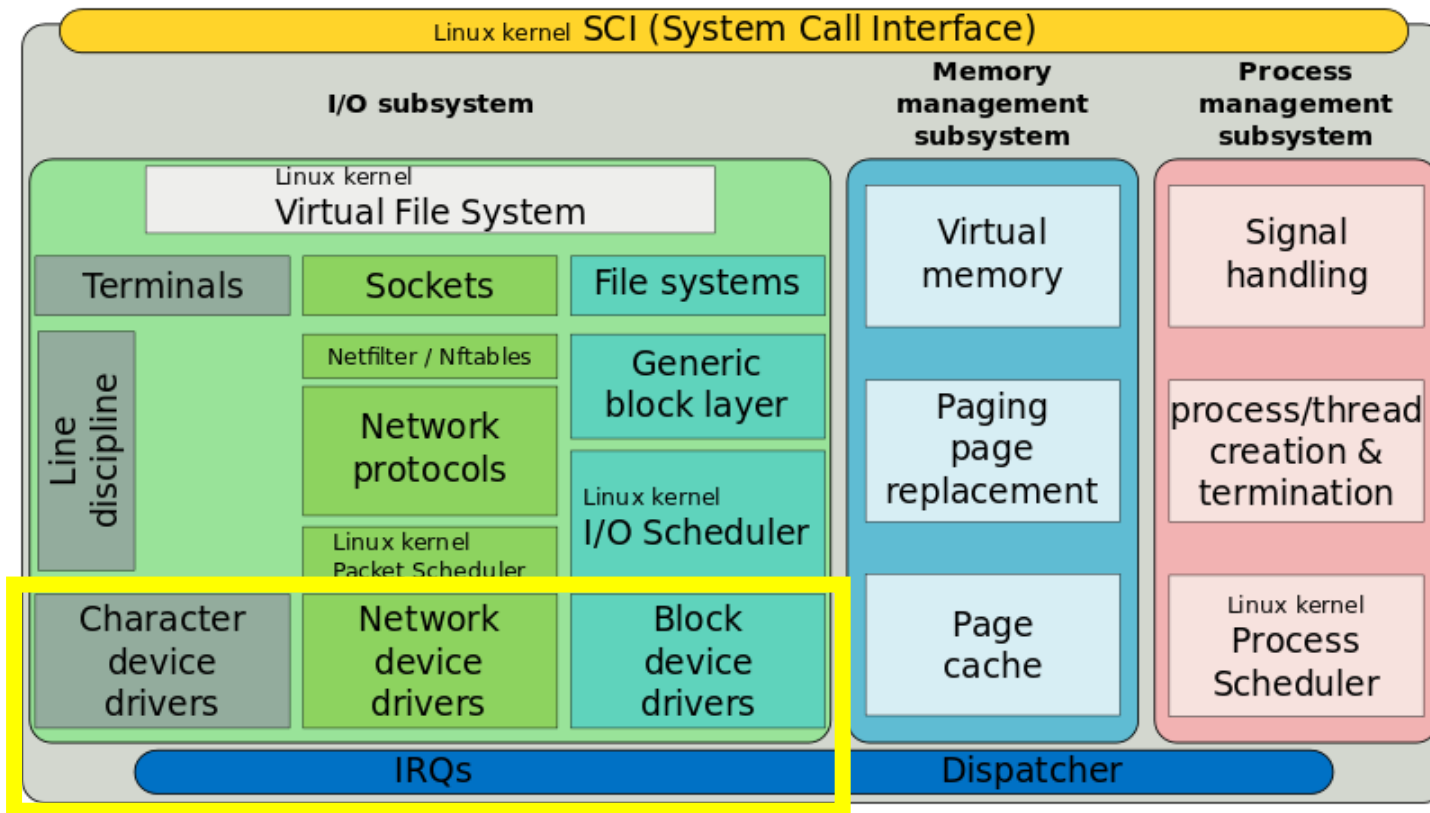
- Address translation
 - All the data user processes give to the kernel comes with virtual addresses
 - Pointers are either going to have to be translated
 - Or memory is going to need to be copied
- Buffering
 - Kernel may need to hold on to a copy of data
 - Especially in asynchronous case
 - When copies are done and how many times is a big kernel efficiency question

Outline

- Abstractions
- **Device I/O layers**
 - Application Layer
 - Kernel I/O Subsystem
 - **Device Driver**
 - **Interrupt Handler**
- Example Driver: Nautilus Character Device
- Example Driver: Temperature Sensor

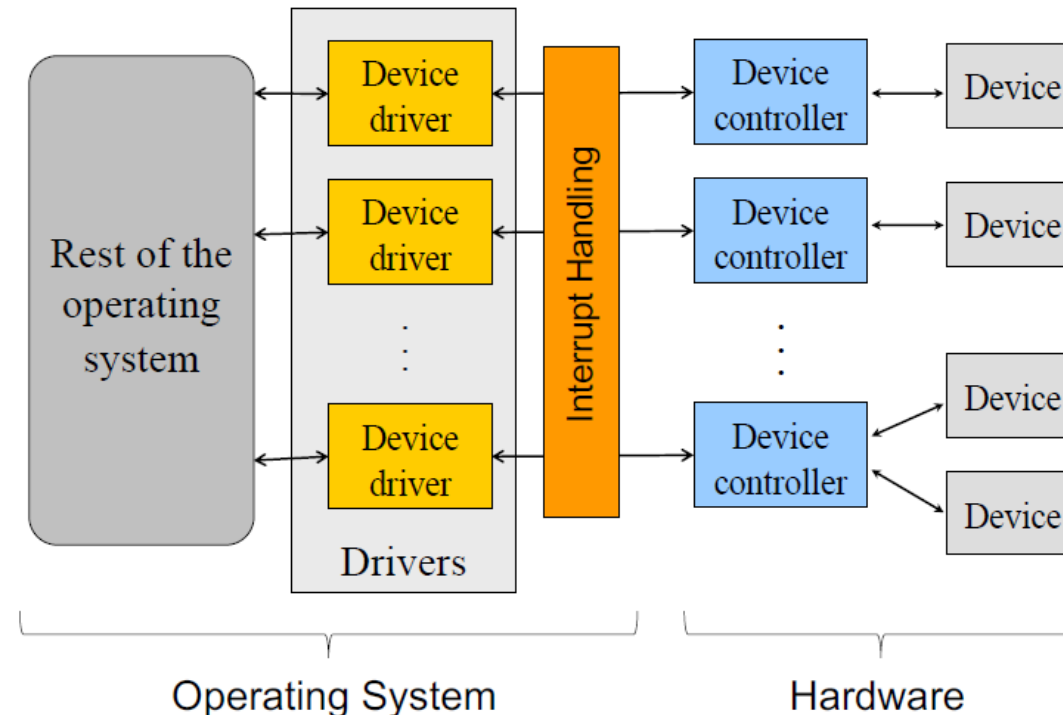
Where we are at in the system

Processes

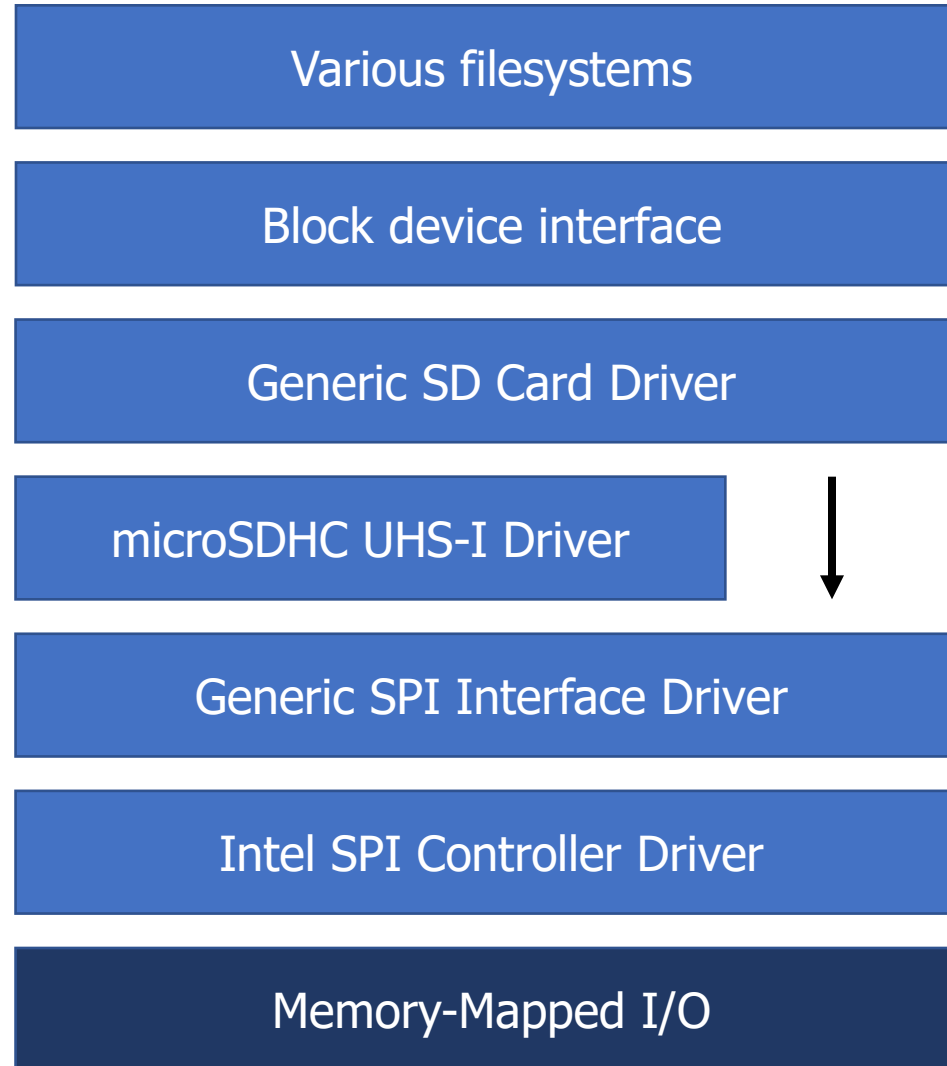


Device drivers

- Device-specific code for communicating with device
 - Supports some interfaces above and below
 - Possibly file syscalls above and memory-mapped I/O below
 - Possibly internal API above and below..
- Examples
 - Specific disk drivers are layered on top of SATA driver
 - Keyboard driver is layered on top of USB driver
 - Ethernet driver has various network interfaces layered above it



Example: possible driver layers for an SD card



Device I/O is handled by device drivers

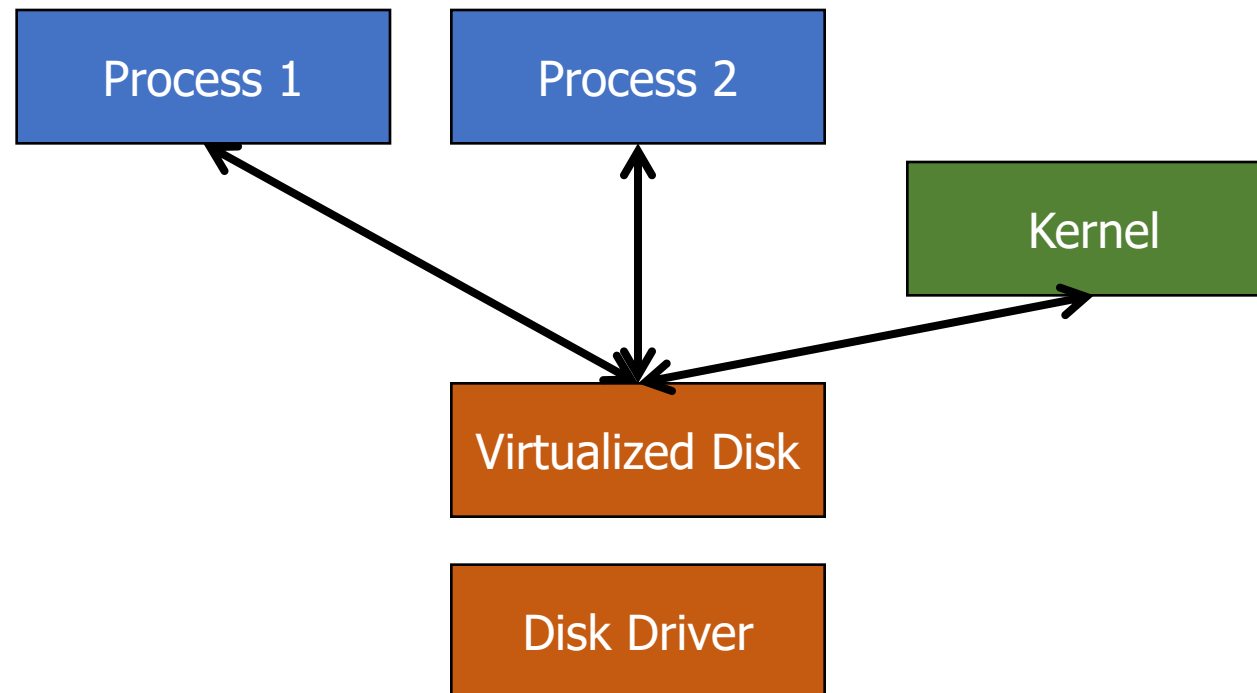
- Communication is up to the hardware
 - Port-mapped I/O or memory-mapped I/O
 - Or function calls to a lower-level driver
- Interaction design is up to the driver (and OS)
 - Programed I/O
 - Synchronous or with interrupts
 - Direct Memory Access
 - Needs hardware support
 - With interrupts

Device drivers are often designed with two “halves”

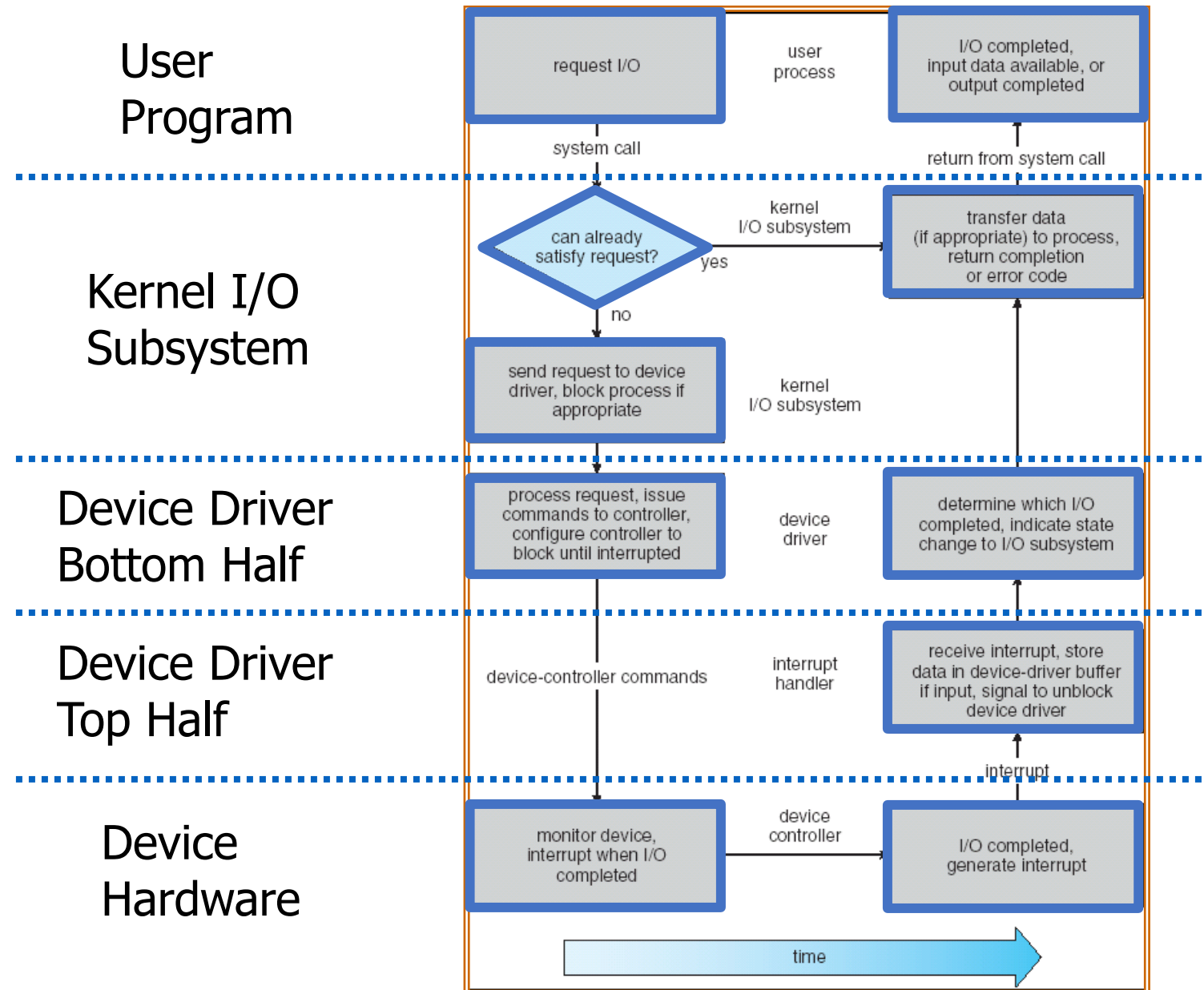
- Top half
 - Interrupt handler
 - Continues next transaction
 - Or signals for bottom half to continue (often with shared variable)
- Bottom half
 - Implements interface that higher layers require
 - Performs logic to start device requests
 - Wait for I/O to be completed
 - Synchronously (blocking) or asynchronously (return to kernel)
 - Handle responses from the device when complete

Virtualizing one device for many users

- Some devices need to be *virtualized*
 - Software that emulates unique devices for each higher level user even though only a single hardware resource actually exists



Life cycle of an I/O request



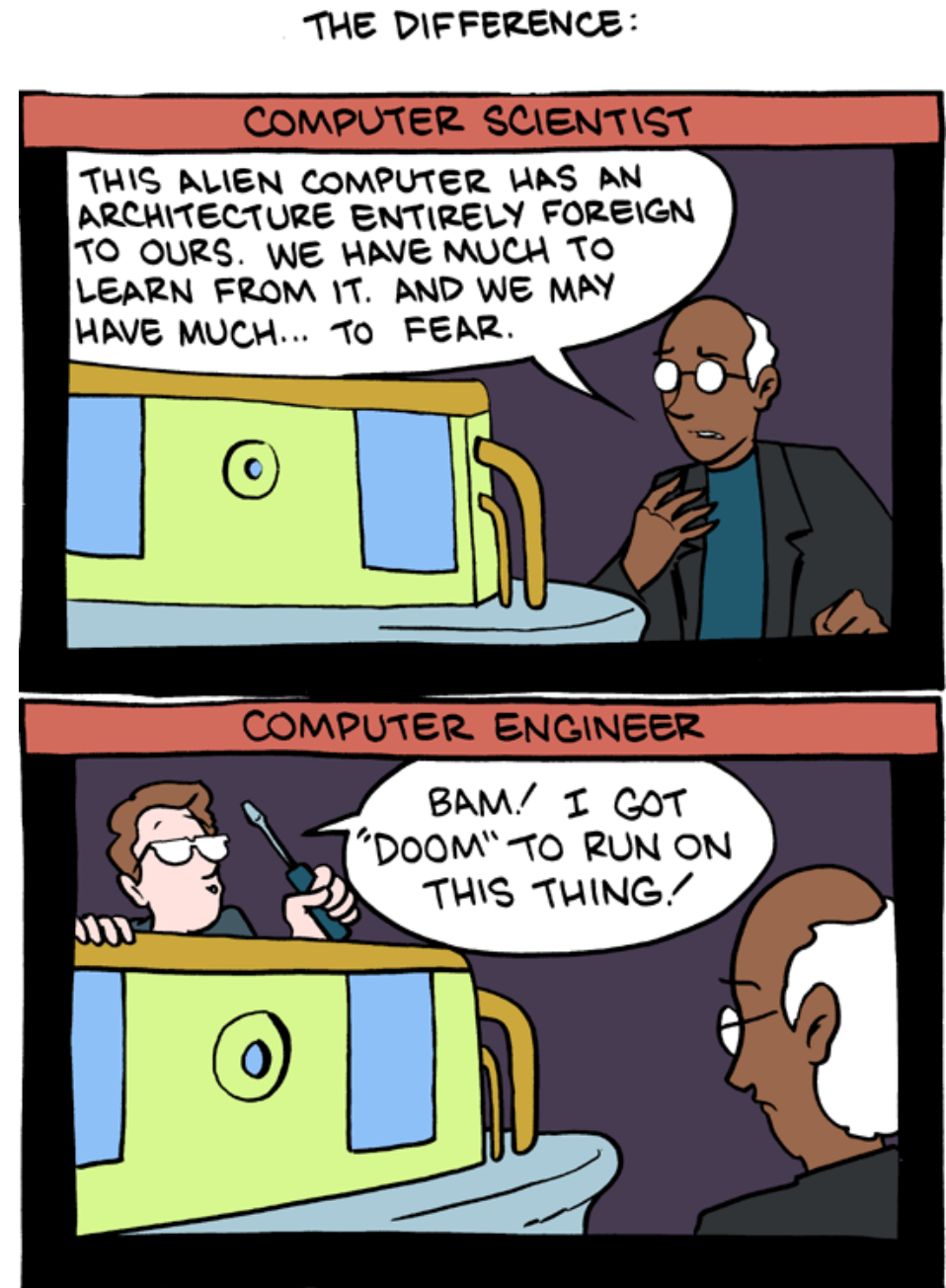
How are devices found anyways?

- At boot, the OS kernel searches for devices attached to it
 - Action is usually called “probe”
 - Starts up drivers for each device it finds
 - A significant amount of time is spent in device discovery
- Run “dmesg” on linux to see printouts from this process
 - Live demo!
 - F25 Moore Timestamps
 - 0.000000 – processor details, RAM discovery
 - 0.044221 – security mitigations
 - 0.599513 – PCI bus discovery
 - 0.636179 – USB discovered
 - 1.134856 – USB devices discovered
 - 1.227143 – RTC accessed
 - 1.384599 – hostname
 - 1.764729 – RAID device started

Break + SMBC webcomic

- Not really relevant to class, just amuses me
- Take a break and reset your brains for a minute

<https://www.smbc-comics.com/comic/2011-02-18>



Outline

- Abstractions
- Device I/O layers
 - Application Layer
 - Kernel I/O Subsystem
 - Device Driver
 - Interrupt Handler
- **Example Driver: Nautilus Character Device**
- Example Driver: Temperature Sensor

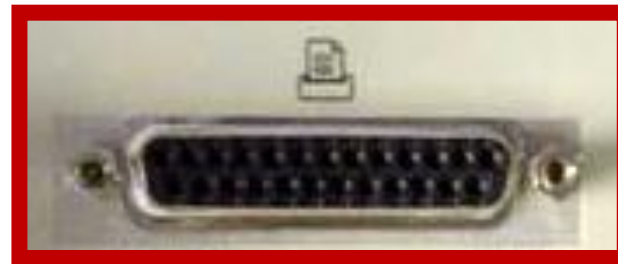
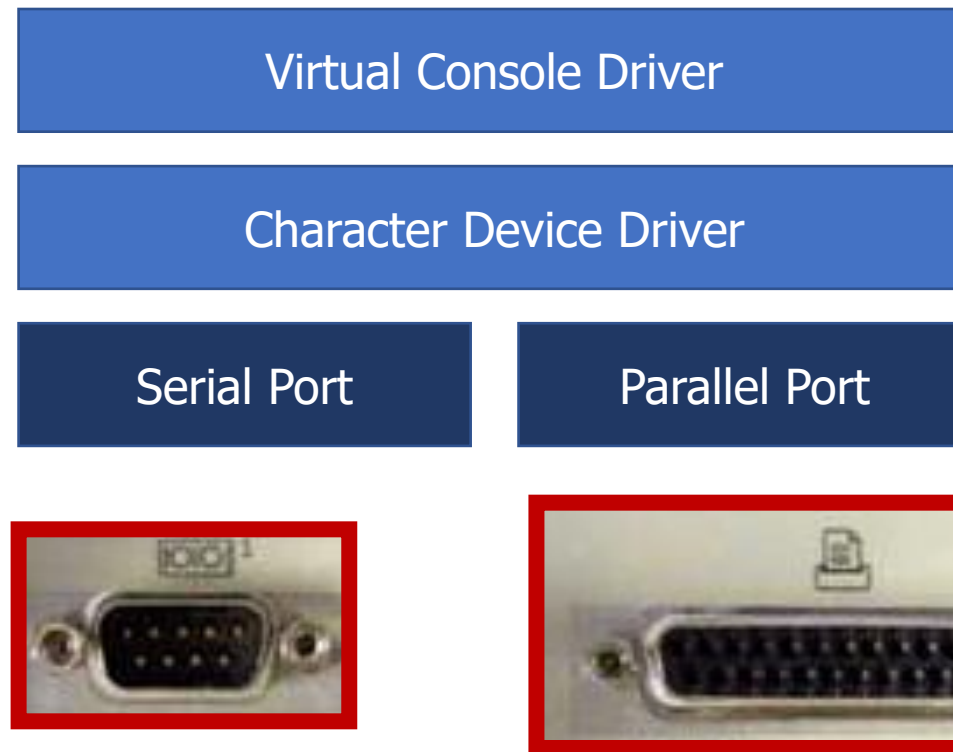
Nautilus kernel

- <https://users.eecs.northwestern.edu/~kch479/nautilus/>
- Small, light-weight kernel for research use
 - All the basic features for getting an x86-64 computer to boot
 - And just about nothing else
- Created by Kyle Hale (Oregon State University) and Peter Dinda (Northwestern)
- Example use case: experiment with virtual memory strategies

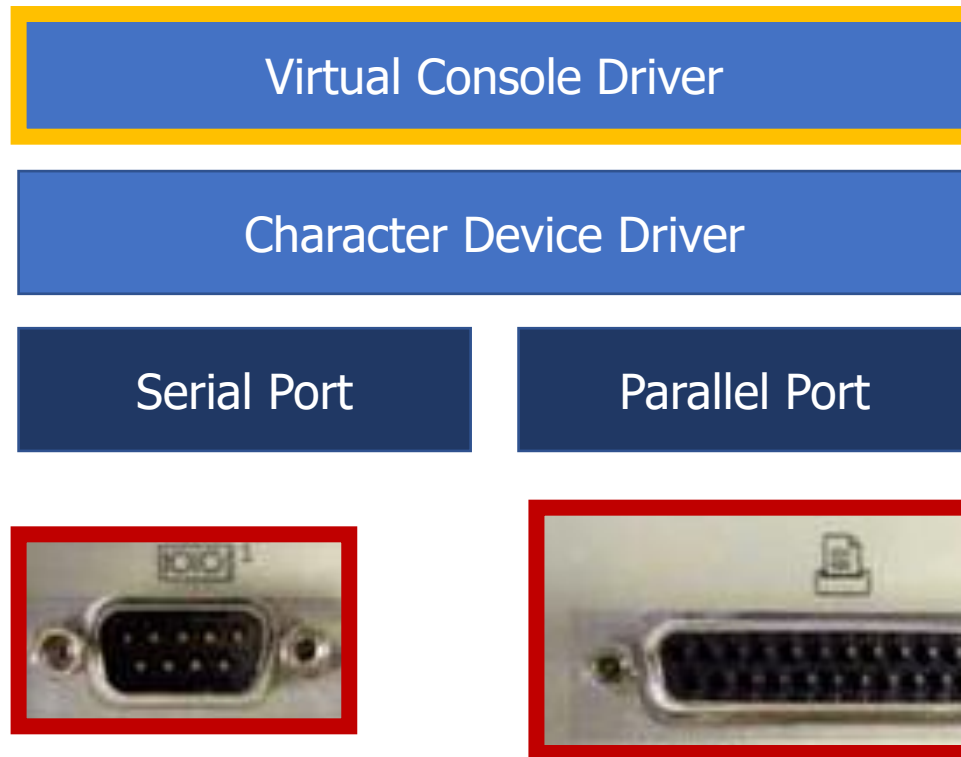
Nautilus character device abstraction

- Character device: a device that can read/write arbitrary characters
 - (as compared to Block devices that must read/write in chunks)
- Nautilus says every character device must have the following:
 - `get_characteristics()` – every device has this, none particularly for chardev
 - `read()` – single byte
 - `write()` – single byte
 - `status()` – determine if device is readable or writeable or both

Layering in Nautilus



Layering in Nautilus



Virtual console

- Allows keyboard input and text output for a user
 - Generally, the basic terminal that you have open
- Could be implemented in all kinds of ways though
 - Example: keyboard input plus printer output
- Any device that can read/write individual characters could act as a console
- So the virtual console just contains a `nk_char_dev`
 - Passed into the virtual console at initialization
 - Could be implemented with any hardware

Virtual console reads and writes to generic char_dev

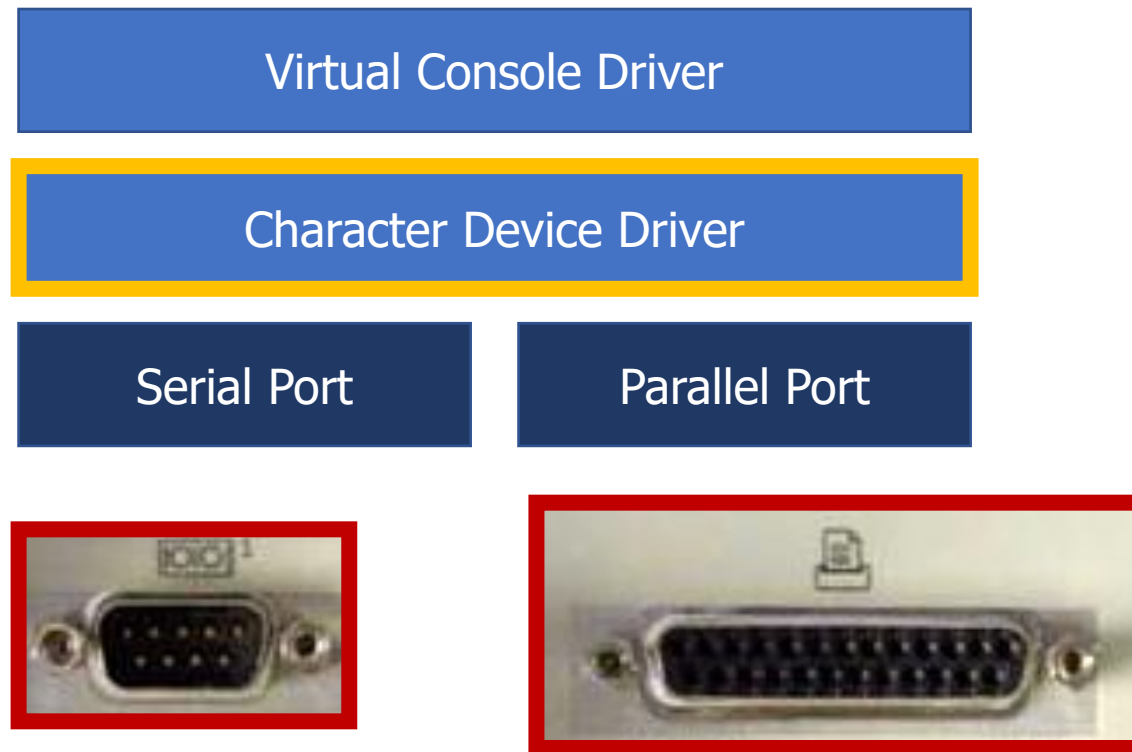
```
char buf[80];  
  
snprintf(buf,80,"\r\n*** Console %s // prev=''1 next=''2 list=''3 ***\r\n",myname);  
char_dev_write_all(c->dev,strlen(buf),buf,NK_DEV_REQ_BLOCKING);
```

- Tries to write an entire string in blocking mode
 - Should not return until the entire string is displayed
- `char_dev_write_all()` is a helper within the virtual console

Virtual Console implements by calling into nk_char_dev

```
static int char_dev write_all(struct nk_char_dev *dev,  
                             uint64_t count,  
                             uint8_t *src,  
                             nk_dev_request_type_t type) {  
  
    uint64_t left, cur;  
    left = count;  
    while (left>0) {  
        cur = nk_char_dev_write(dev, left, &(src[count-left]), type);  
        if (cur == -1ULL) {  
            return -1;  
        } else {  
            left-=cur;  
        }  
    }  
  
    return 0;  
}
```

Layering in Nautilus



Each nk_char_dev holds an interface of function pointers

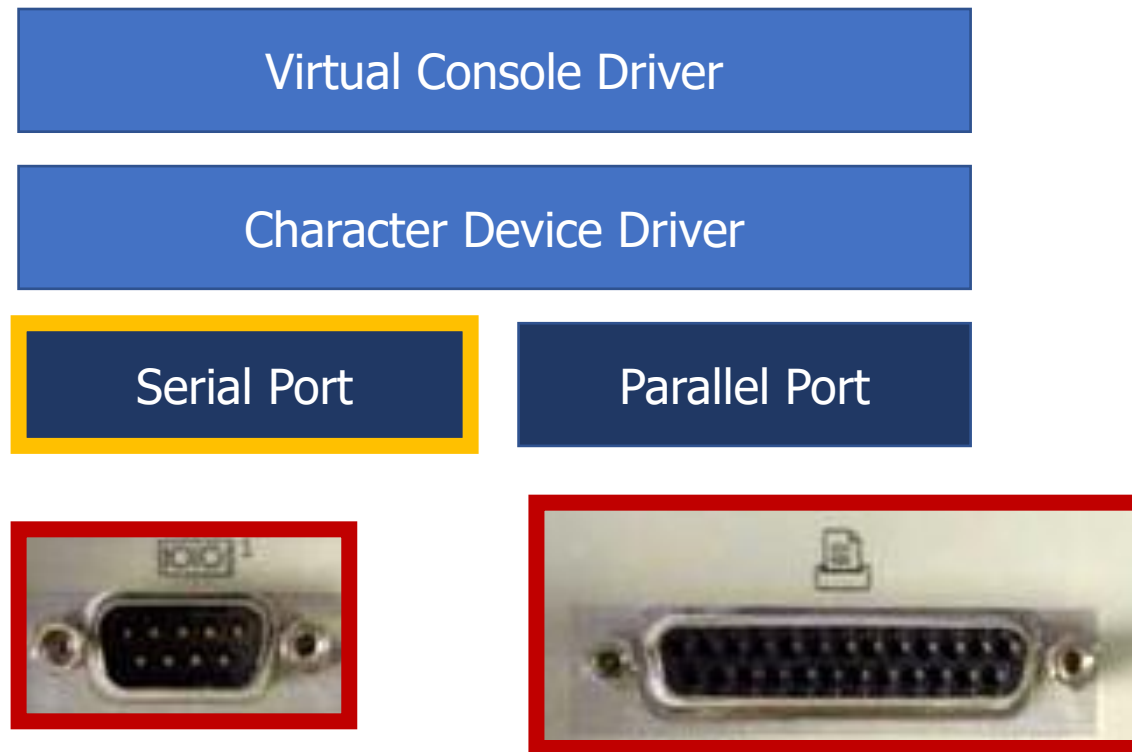
```
struct nk_char_dev_int {  
    // this must be first so it derives cleanly  
    // from nk_dev_int  
    struct nk_dev_int dev_int;  
  
    // chardev-specific interface - set to zero if not available  
    // either succeeds (returns zero) or fails (returns -1)  
    int (*get_characteristics)(void *state, struct nk_char_dev_characteristics *c);  
  
    // returns 1 on success, 0 for would block, -1 for error  
    // must be non-blocking  
    int (*read)(void *state, uint8_t *dest);  
    int (*write)(void *state, uint8_t *src);  
  
    // returns whether device is currently readable or writeable or both  
    // or in error state  
#define NK_CHARDEV_READABLE 1  
#define NK_CHARDEV_WRITEABLE 2  
#define NK_CHARDEV_ERROR 4  
    int (*status)(void *state);  
};
```

Simplified nk_char_dev_write: calls write() operation

```
uint64_t nk_char_dev_write(struct nk_char_dev *dev,
                           uint64_t count,
                           uint8_t *src,
                           nk_dev_request_type_t type){
    struct nk_dev *d = (struct nk_dev *)&(dev->dev);
    struct nk_char_dev_int *di = (struct nk_char_dev_int *)(d->interface);

    uint64_t num=0;
    int err;
    while (num<count) {
        err = di->write(d->state,src);
        if (err < 0) {
            return -1;
        } else if (err==0) {
            nk_dev_wait((struct nk_dev *)dev, is_writeable, dev);
        } else {
            num++;
            src++;
        }
    }
    return num;
}
```

Layering in Nautilus



A serial device implements the nk_char_dev operations

```
static struct nk_char_dev_int chardevops = {  
    .get_characteristics = serial_do_get_characteristics,  
    .read = serial_do_read,  
    .write = serial_do_write,  
    .status = serial_do_status  
};
```

- Serial device implements all of those operations
- When you create a serial device, you actually make an `nk_char_dev` and initialize it with a `chardevops`
 - All of the generic device operations call into the actual serial device

Simplified Serial device: pushes data into a queue

```
static int serial_do_write(void *state, uint8_t *src) {  
    struct serial_state *s = (struct serial_state *)state;  
  
    int flags;  
  
    flags = spin_lock_irq_save(&s->output_lock);  
  
    serial_output_push(s, *src);  
  
    kick_output(s);  
    spin_unlock_irq_restore(&s->output_lock, flags);  
    return 1;  
}
```

Serial queue operation

- Whenever a write comes in, we push data byte into a queue
 - Serial output goes slowly, so many bytes could be queued up
- Then we enable interrupts and write the first byte to the MMIO register

```
*(volatile uint8_t*)(s->addr + offset) = val;
```

- Then when an interrupt comes in, we pop the next byte from the queue and write it to the MMIO register
 - Repeats until the queue is empty

Example from last lecture of writing to a temperature sensor

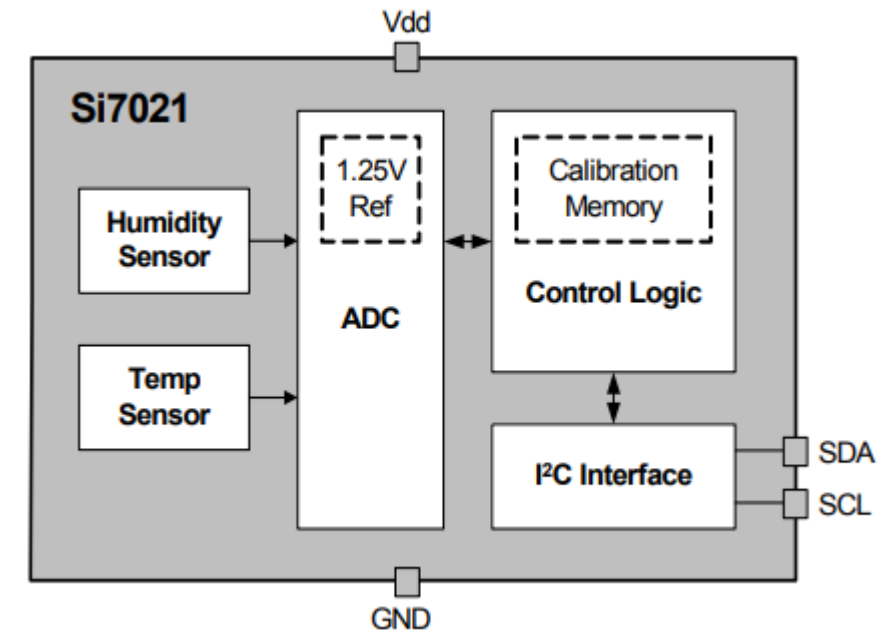
```
*(uint32_t*)(0x4000C000) = 1;
```

Outline

- Abstractions
- Device I/O layers
 - Application Layer
 - Kernel I/O Subsystem
 - Device Driver
 - Interrupt Handler
- Example Driver: Nautilus Character Device
- **Example Driver: Temperature Sensor**

Si7021 temperature and humidity sensor

- Popular on embedded devices
 - Also has a Linux driver!
- Connects to computer over I²C bus
 - Two-wire, 100 Kbps low-power bus
 - Like any other bus
 - Takes an address
 - Whether it's a read or write transaction
 - And an amount of data



- <https://www.silabs.com/documents/public/data-sheets/Si7021-A20.pdf>

How do we make it do anything?

- Typically with I²C devices, you write a 1-2 byte command
 - Then you read the data in the next transaction
 - Commands are found in the datasheet

Table 11. I²C Command Table

Command Description	Command Code
Measure Relative Humidity, Hold Master Mode	0xE5
Measure Relative Humidity, No Hold Master Mode	0xE5
Measure Temperature, Hold Master Mode	0xE3
Measure Temperature, No Hold Master Mode	0xF3
Read Temperature Value from Previous RH Measurement	0xE0
Reset	0xFE
Write RH/T User Register 1	0xE6
Read RH/T User Register 1	0xE7
Write Heater Control Register	0x51
Read Heater Control Register	0x11
Read Electronic ID 1st Byte	0xFA 0x0F
Read Electronic ID 2nd Byte	0xFC 0xC9
Read Firmware Revision	0x84 0xB8

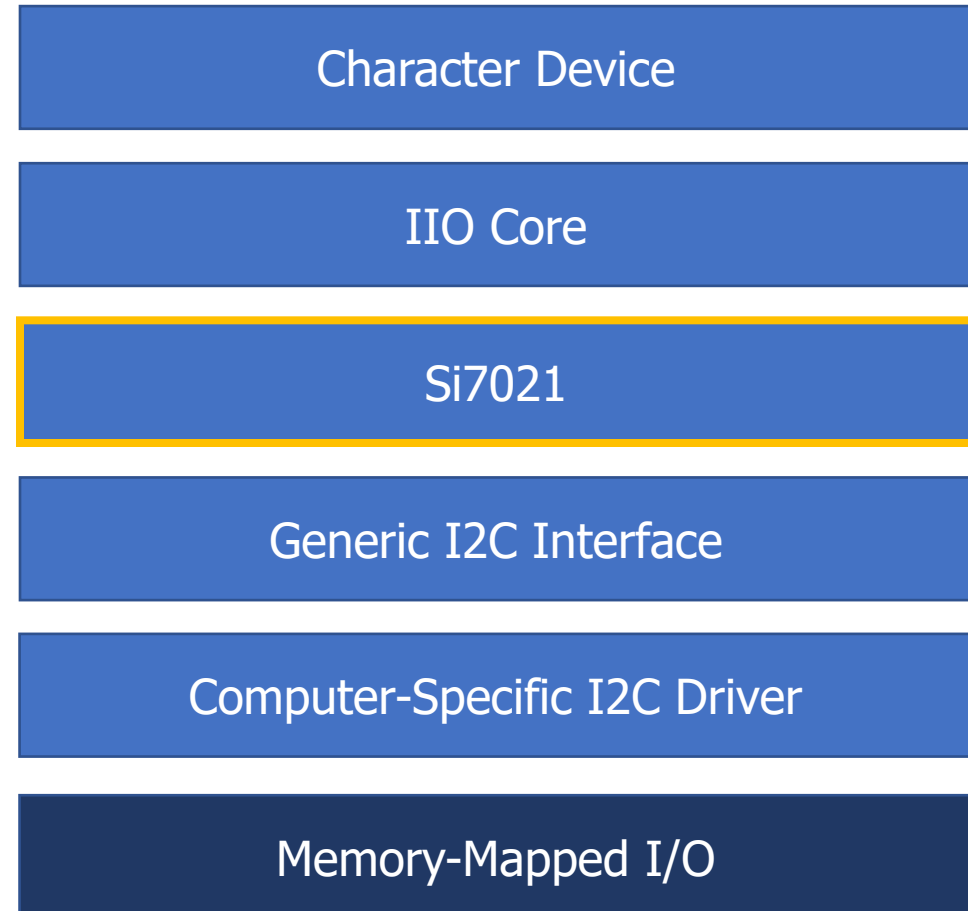
What will the driver look like?

- Layer below it will be I²C controller (function calls)
- In the Temperature driver we need to
 - See what the request from the layer above is
 - Perform an I²C write transaction with a command byte (0xE3)
 - Wait until data is ready
 - Perform an I²C read transaction to get the data
 - Translate the data into meaningful units

$$\text{Temperature (}^{\circ}\text{C)} = \frac{175.72 * \text{Temp_Code}}{65536} - 46.85$$

What are the driver layers going to be?

- In Linux, some sensors are connected through the Industrial I/O subsystem (IIO)
 - Handles sensor data generically
 - Get raw sample
 - Get scaling value
 - Get offset value
- Lower layers could change and everything would still work
 - USB->I2C converter for example
 - Or a totally different sensor



Demo: Linux device driver code for Si7021

<https://github.com/torvalds/linux/blob/master/drivers/iio/humidity/si7020.c>

- Linux source code is all on Github!

But if you want to explore Linux code, a better link is:

<https://elixir.bootlin.com/linux/latest/source/drivers/iio/humidity/si7020.c>

- Creates linked databases for function calls and variable types
 - Lists where it is defined
 - Lists where it is used
- Makes it easy to hop up and down layers

OSes can make design choices about drivers

- Interface does not have to be like a file
 - For example: could have a set of unique syscalls for device interactions
- Asynchronous model could be enforced
 - Must register callback handlers with lower layer to get response
- Tock embedded operating system does both of these
 - <https://www.tockos.org/>

Demo: Tock device driver code for Si7021

<https://github.com/tock/tock/blob/master/capsules/extra/src/si7021.rs>

Outline

- Abstractions
- Device I/O layers
 - Application Layer
 - Kernel I/O Subsystem
 - Device Driver
 - Interrupt Handler
- Example Driver: Nautilus Character Device
- Example Driver: Temperature Sensor