Lecture 15: Device Drivers

CS343 – Operating Systems

Branden Ghena – Fall 2020

Some slides borrowed from:
Stephen Tarzia (Northwestern), Jaswinder Pal Singh (Princeton), and UC Berkeley CS162

Today’s Goals

- Explore how software for device I/O is architected.
- Discuss OS considerations at multiple software layers.
- Investigate an example device driver.

Outline

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

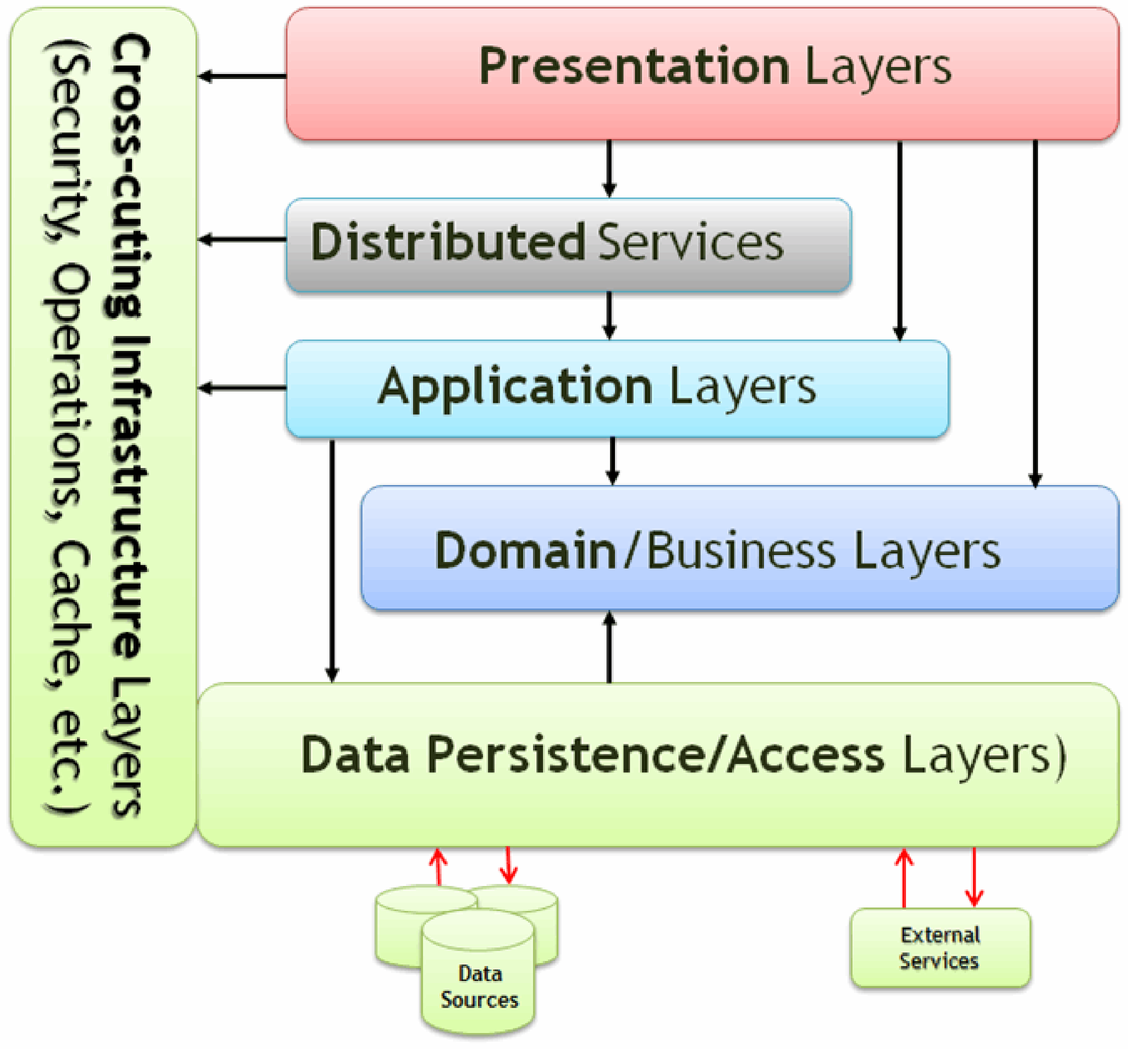
Writing software to manage devices

- Kernel software for managing a device is a *device driver*
  - 70% of Linux code is device drivers
  - 15.3 Million lines of source code

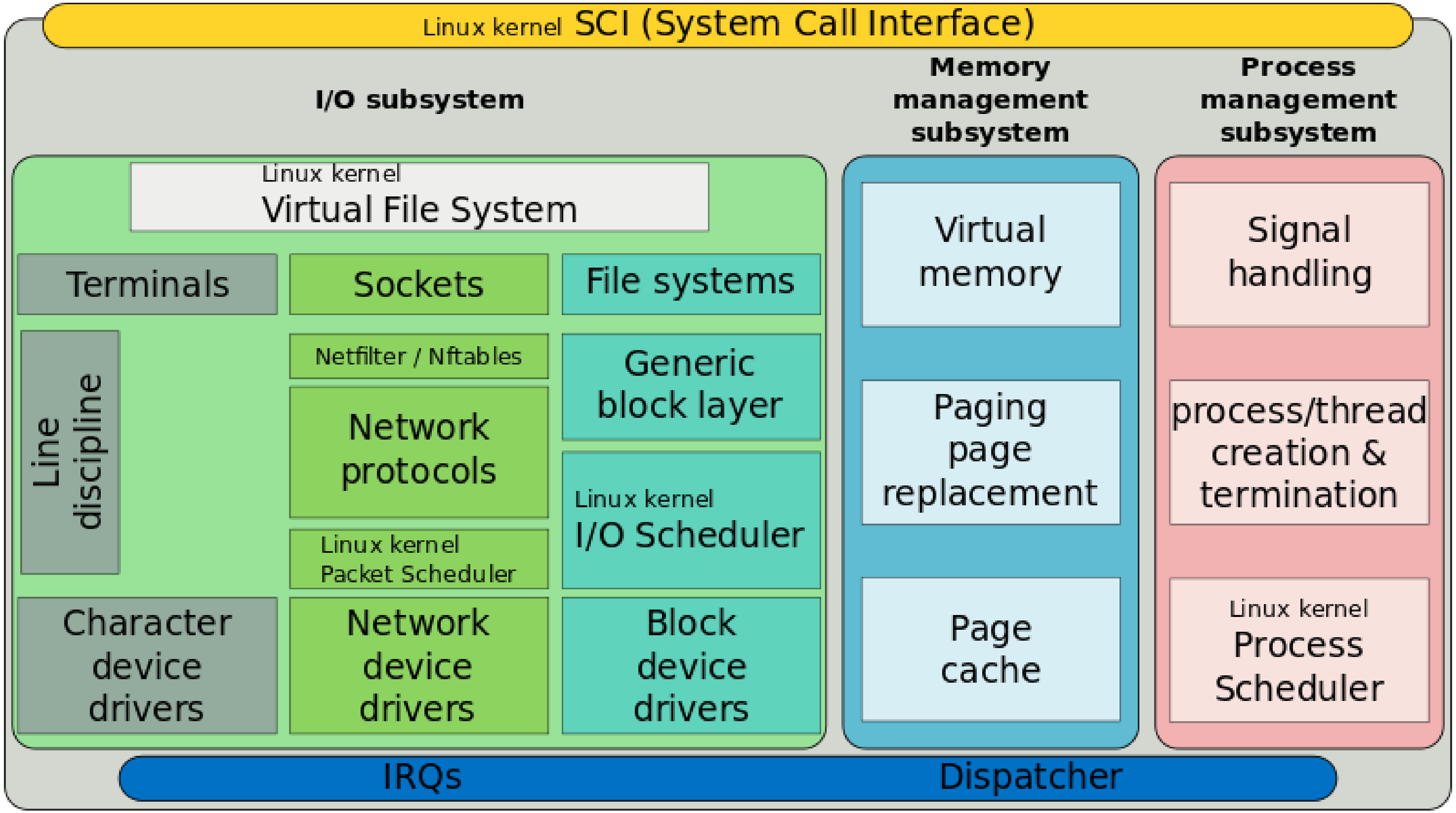
- Big challenge for device drivers
  - How do we enable interactions with so many varied devices?
    - Need abstractions to allow software to interact with them easily
    - Need mechanisms to reuse a lot of code for commonalities

General software abstractions

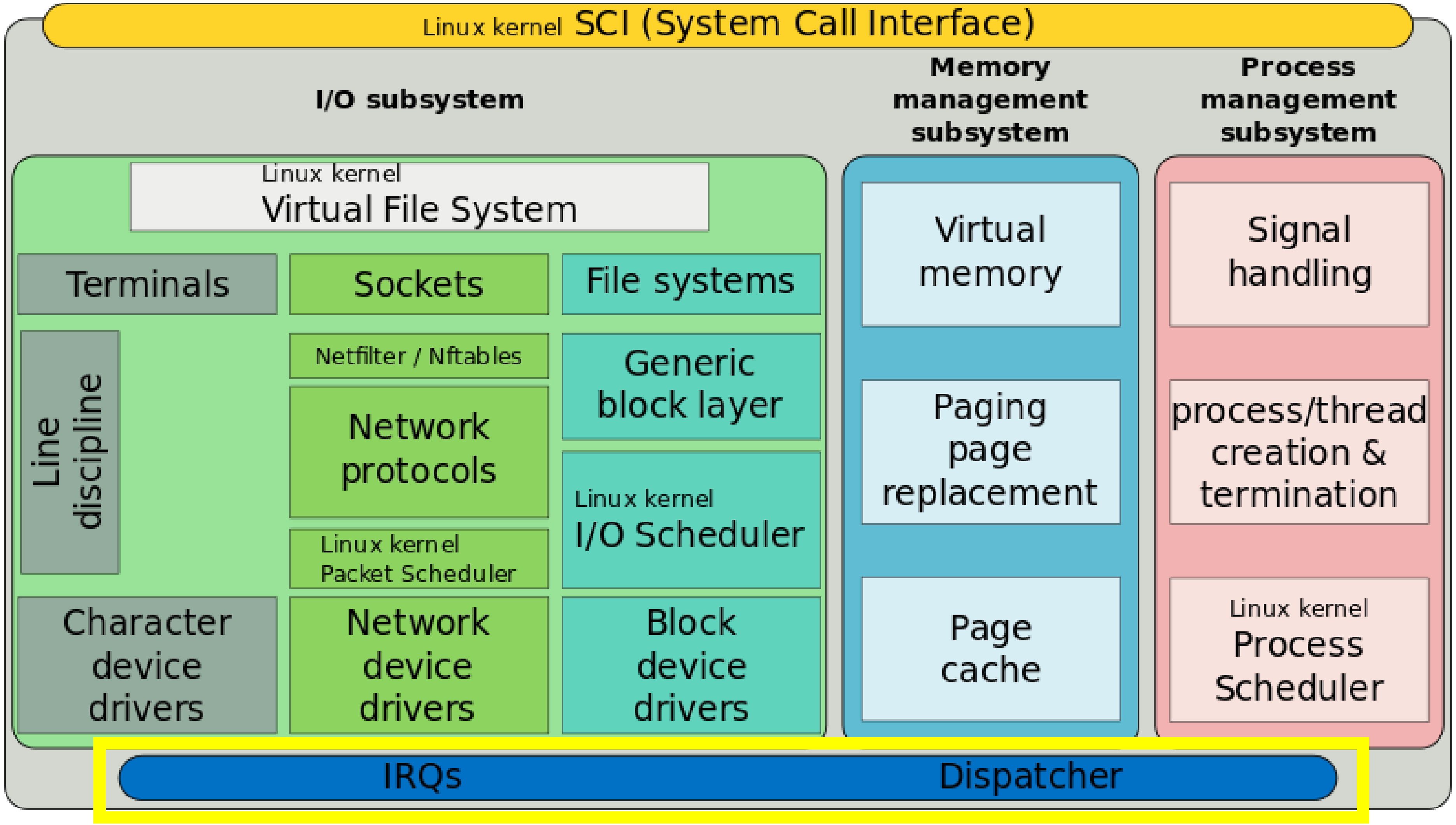
- When building large software projects, we like to define layers of code
  - Makes it clear what is handled where
  - Enables swapping out implementations when desired



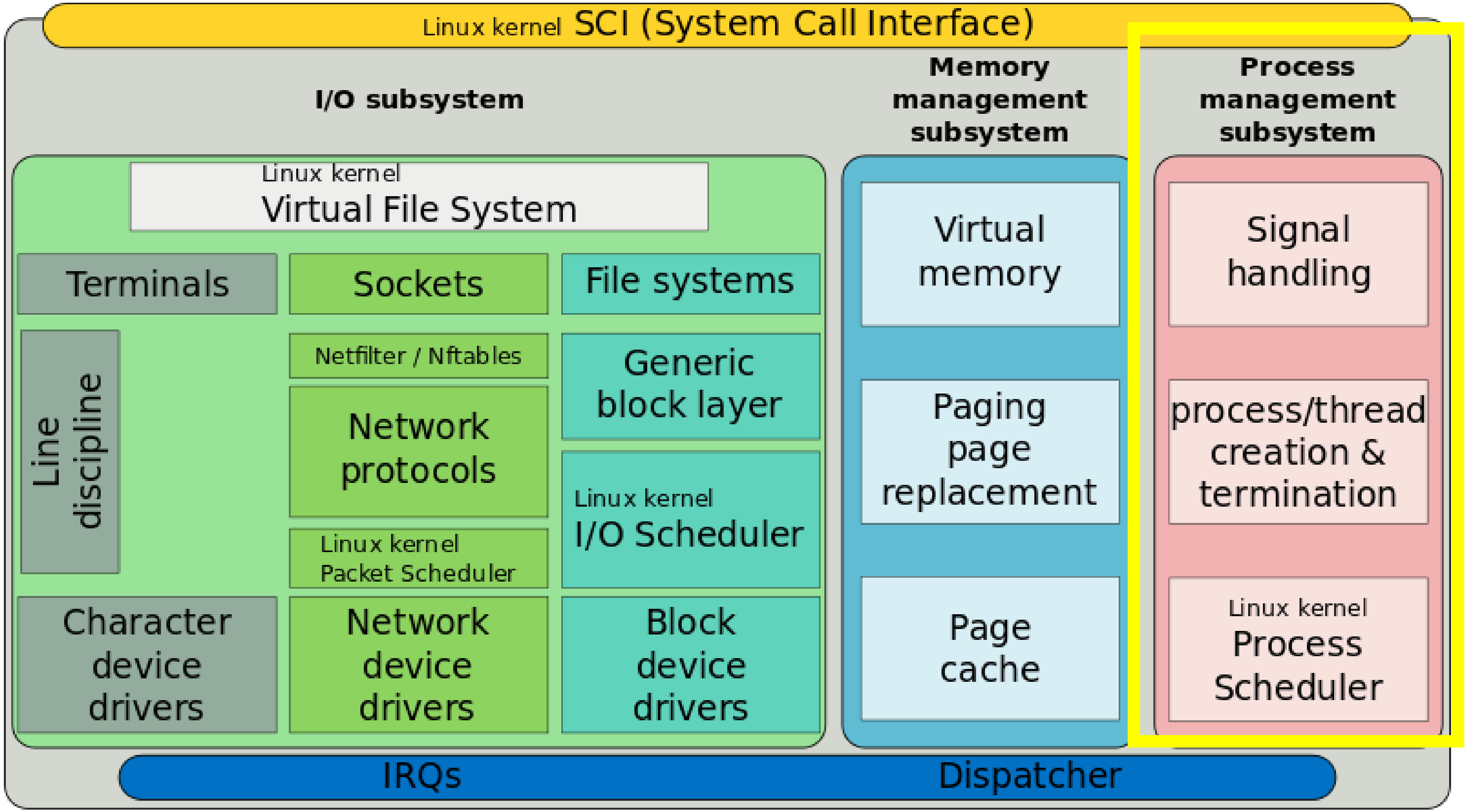
Linux Kernel Layering



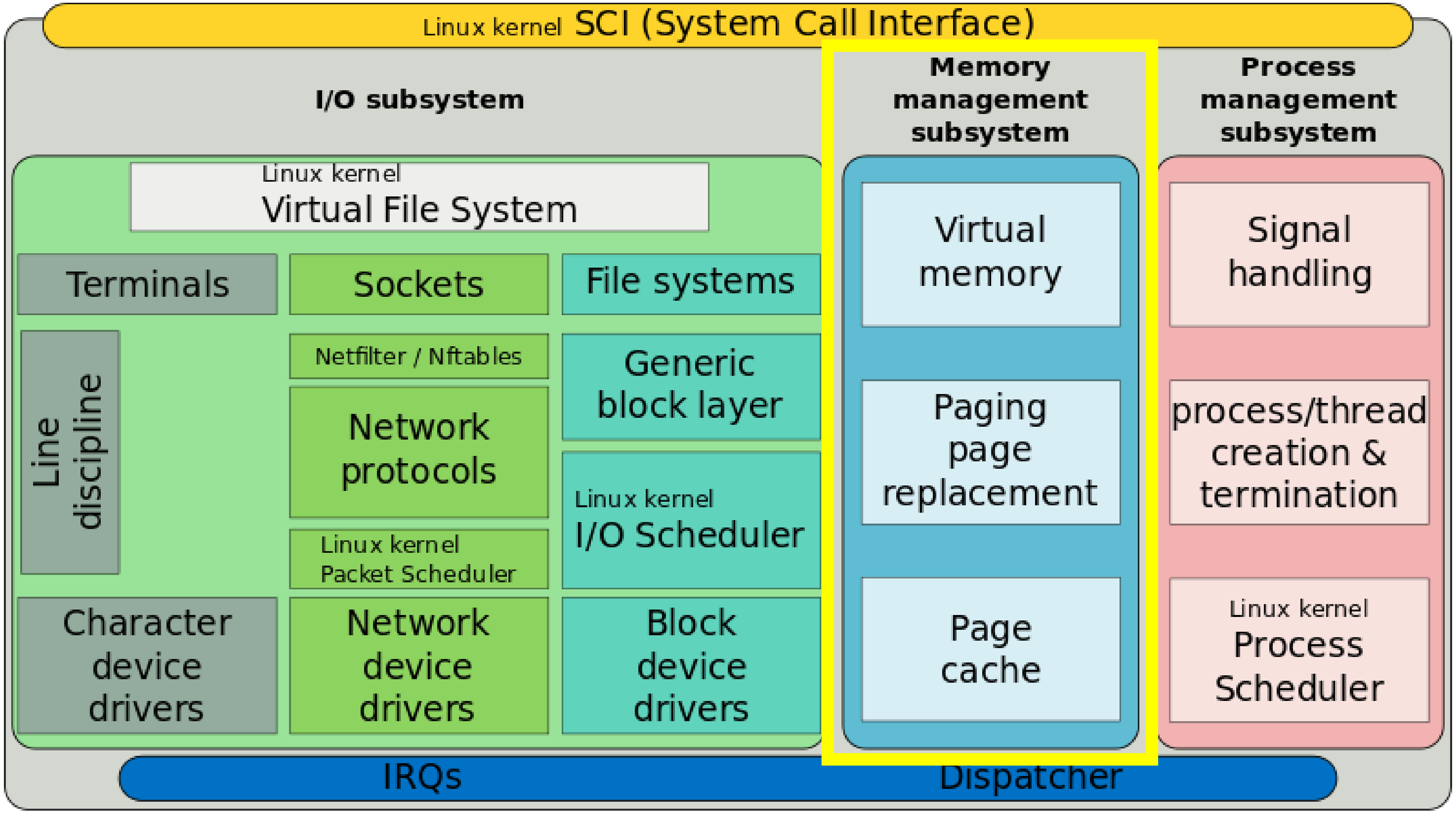
Linux Kernel Layering



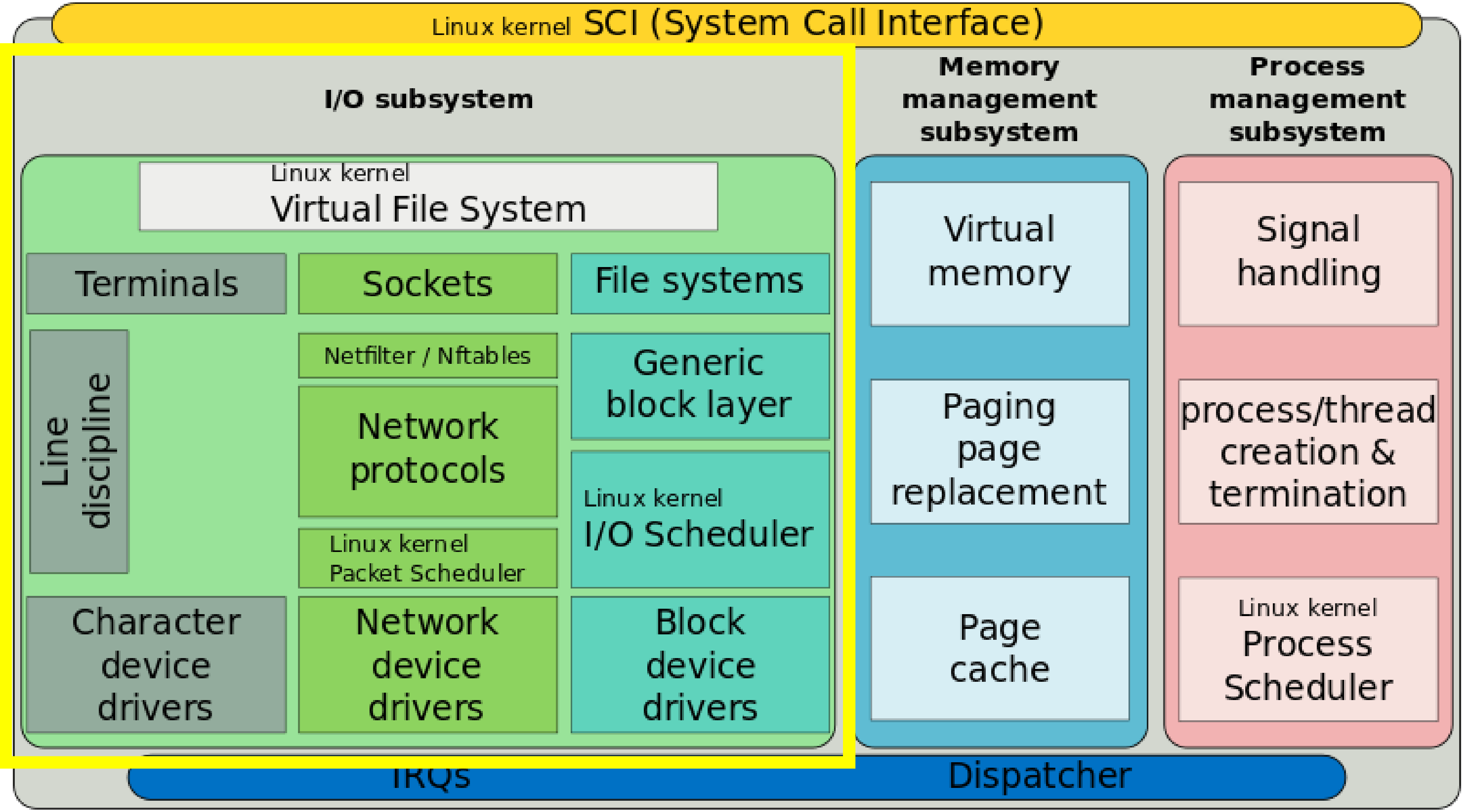
Linux Kernel Layering



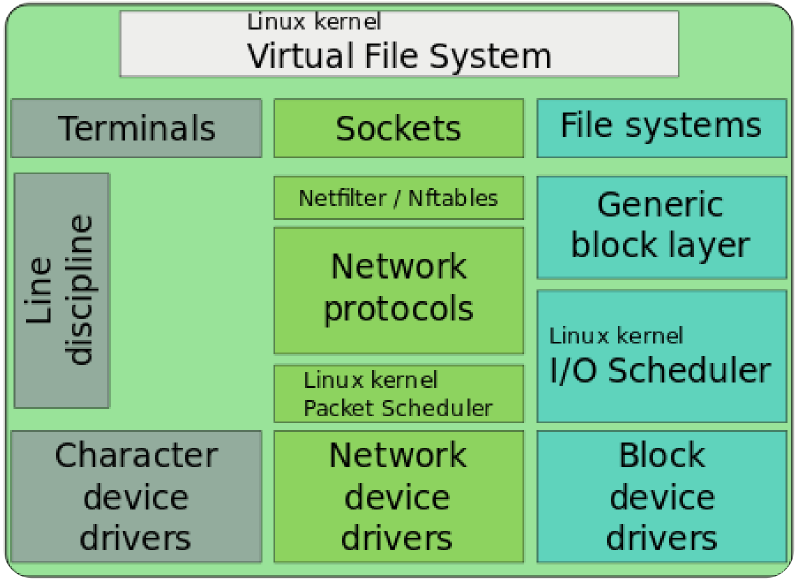
Linux Kernel Layering



Linux Kernel Layering

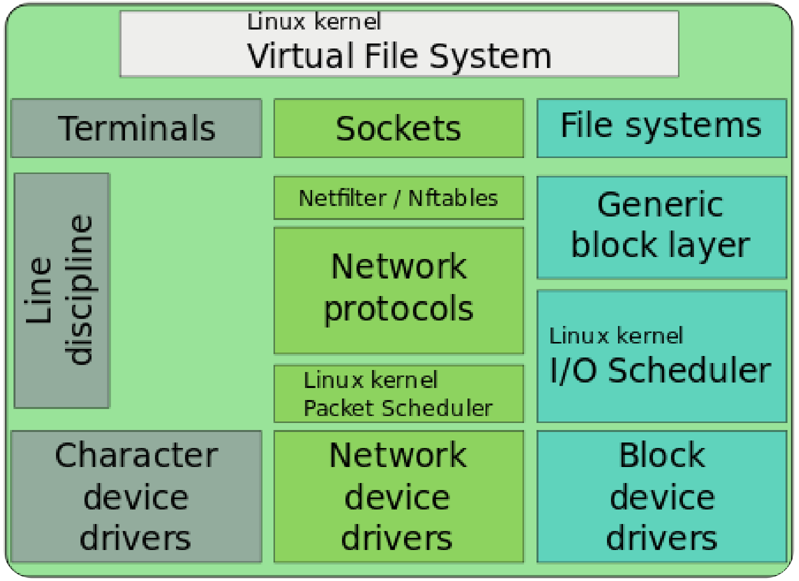


Abstraction: everything is a file!



- 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

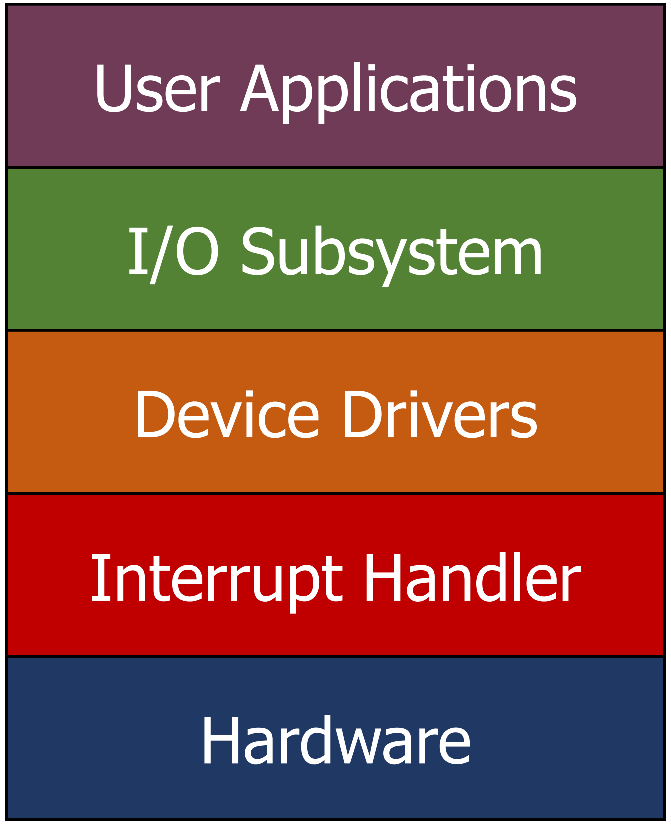
Linux device classes



- 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

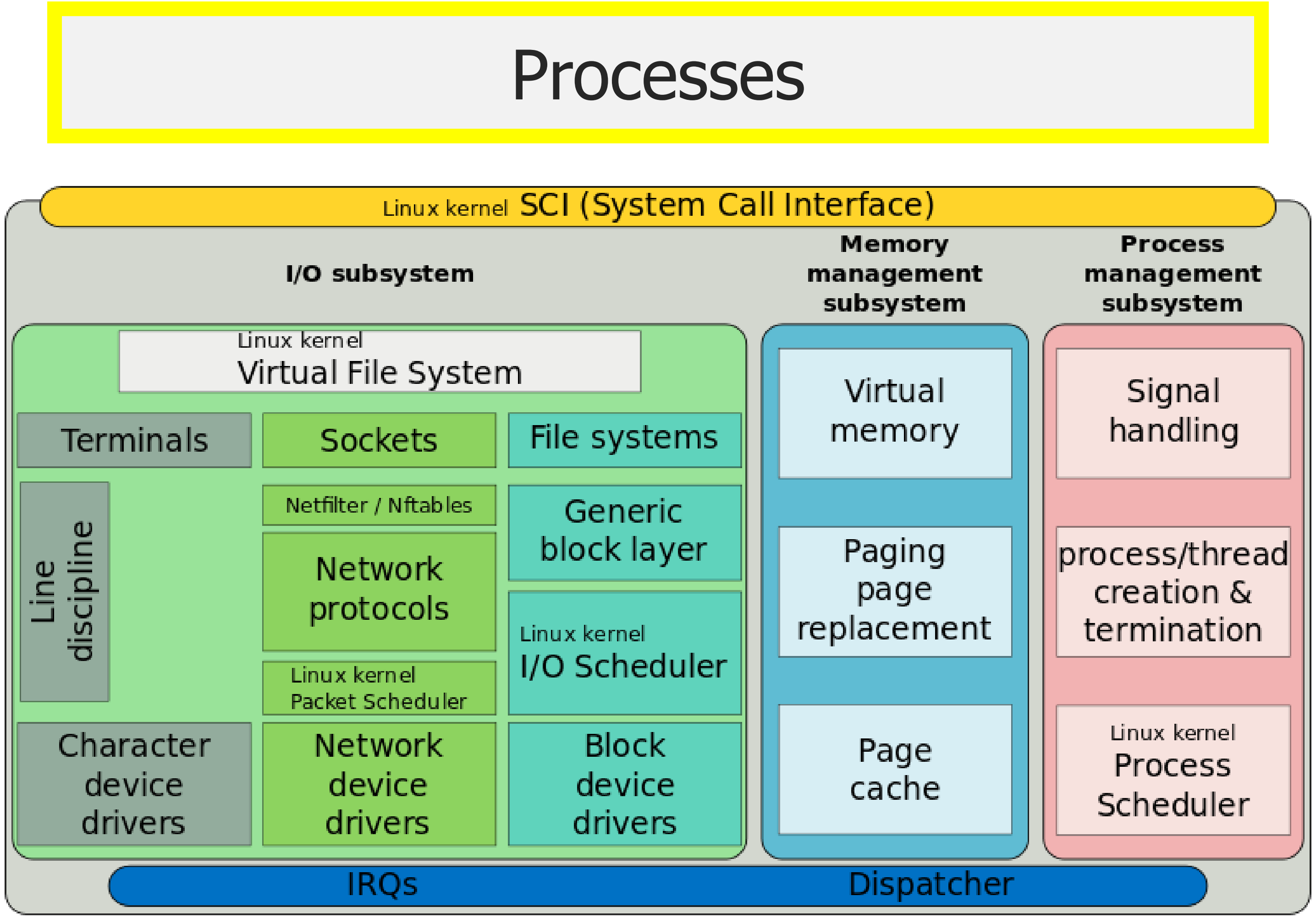
- User applications
  - Do useful things
- I/O subsystem
  - Receive syscalls, route to device drivers
- Device drivers
  - Translate application requests into device interactions
- Interrupt Handler
  - Receive events from hardware
- Hardware
  - Do useful things

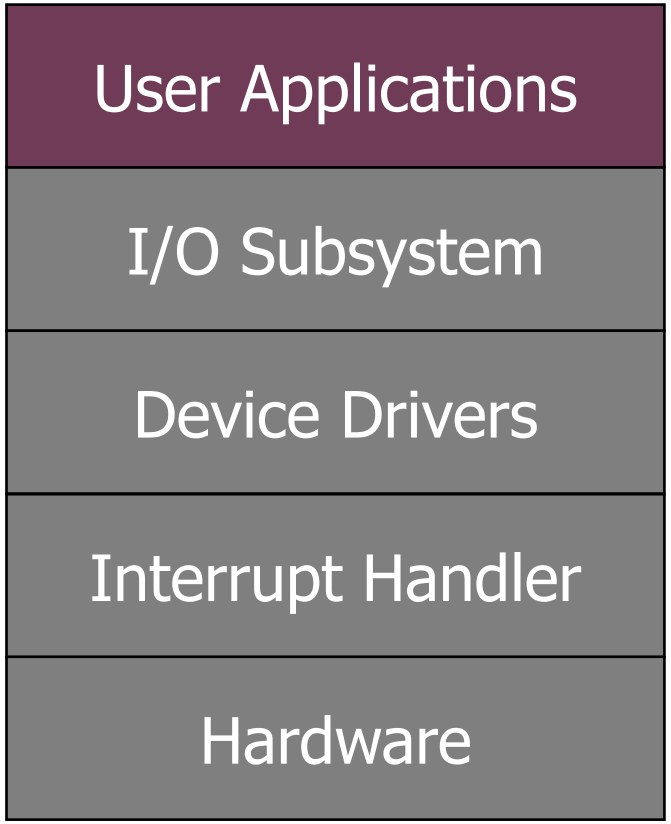


Outline

- Abstractions
- **Device I/O layers**
  - **Application Layer**
  - Kernel I/O Subsystem
  - Device Driver
  - Interrupt Handler
- Example Driver: 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
dmmidi           kvm        mapper        rtc     tty       tty21  tty35  tty49  tty62  ttyS17     ttyS30  vcs2     vcsu3
```

Interacting with devices

- 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

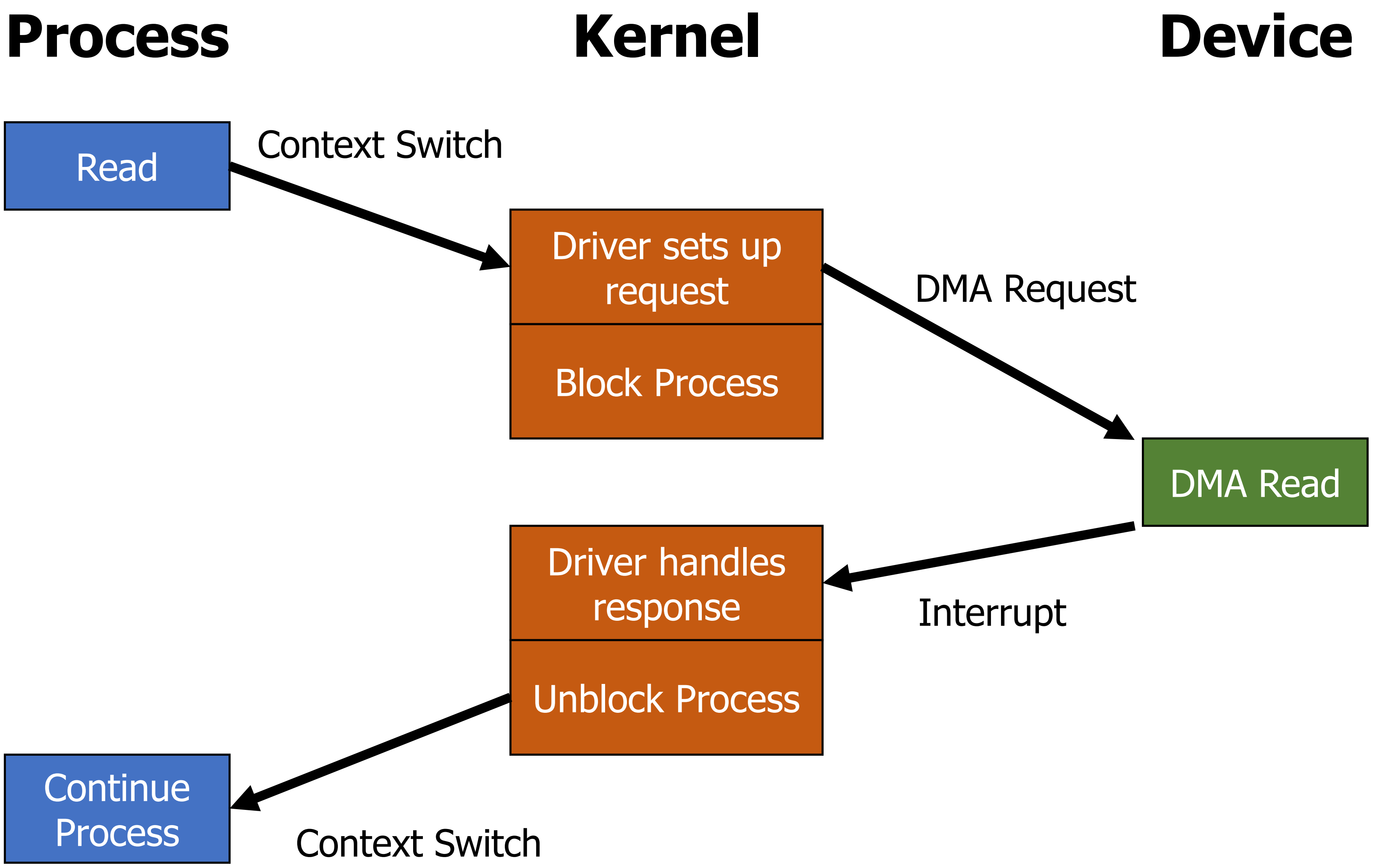
- ioctl – I/O Control
  - `int ioctl(int fd, unsigned long request, ...);`
- Request number followed by an arbitrary list of arguments
  - "request" may be broken in fields: command, size, direction, etc.
- Catch-all for device operations that don't fit into file I/O model
  - Combine magic numbers to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

Asynchronous I/O operations

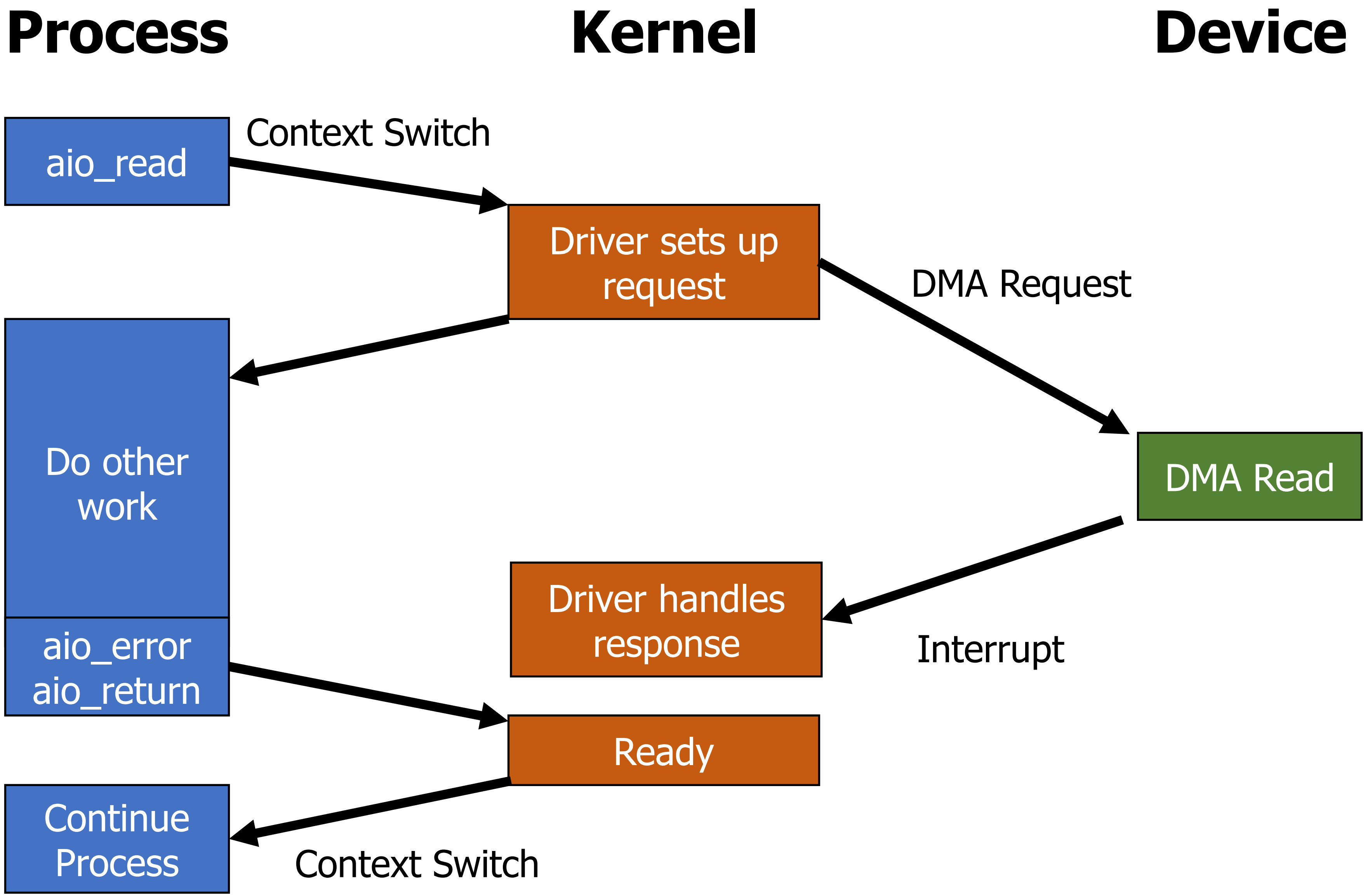
- Previous examples were all synchronous I/O calls
  - Read/Write will block process until complete
  - Easy to use, but not always most efficient method

- 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

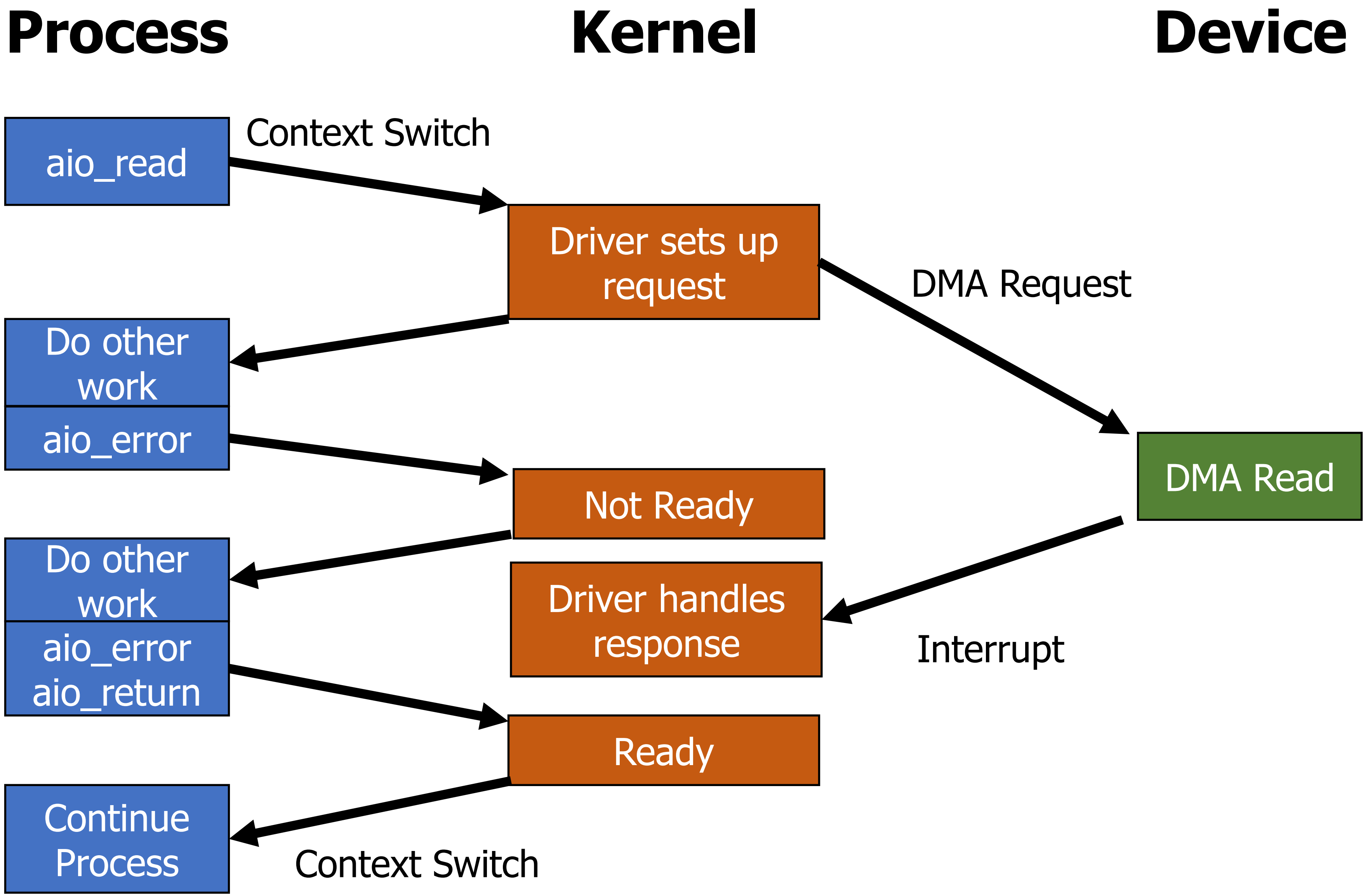
Synchronous blocking read example



Asynchronous read example



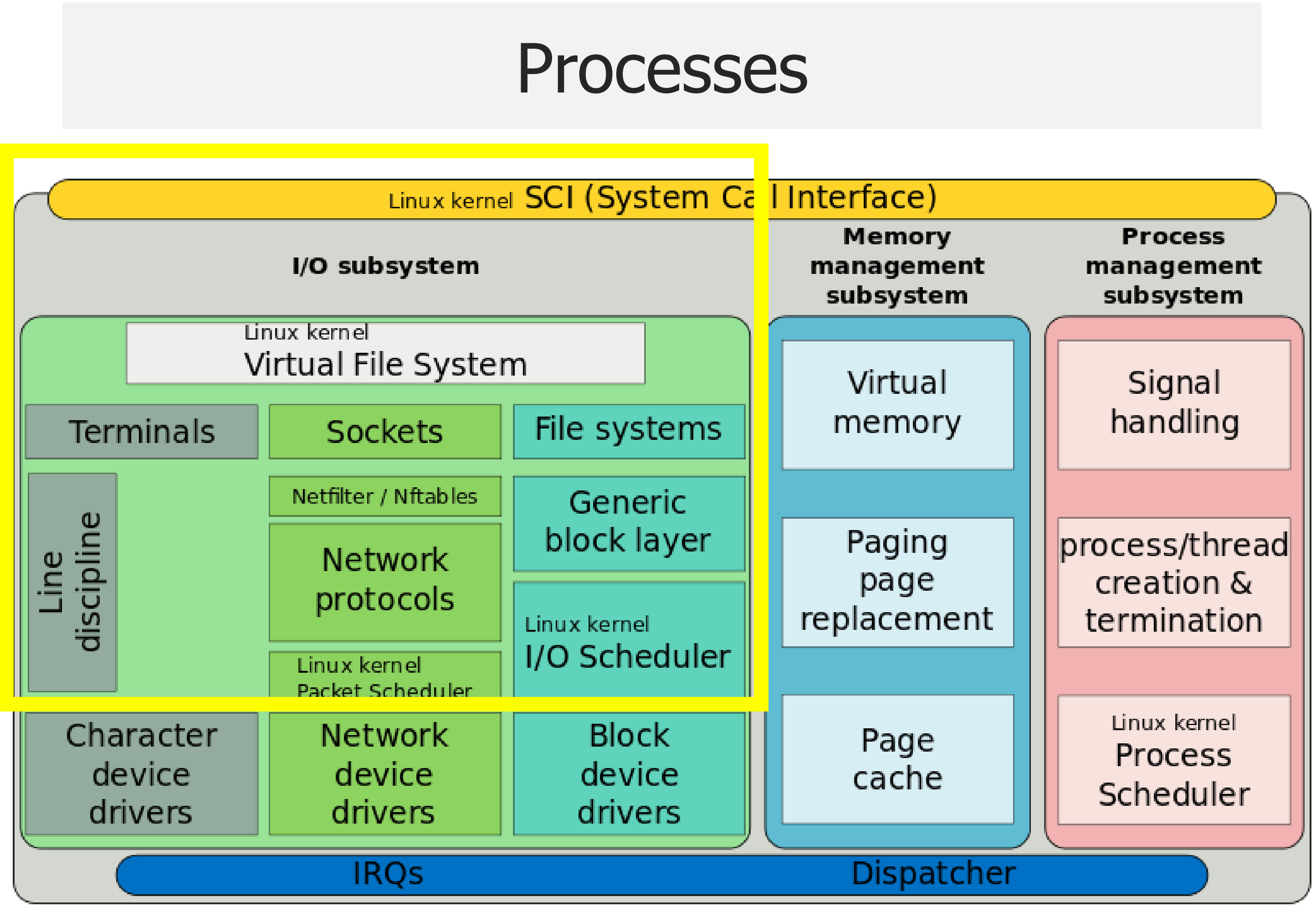
Asynchronous read example with early request

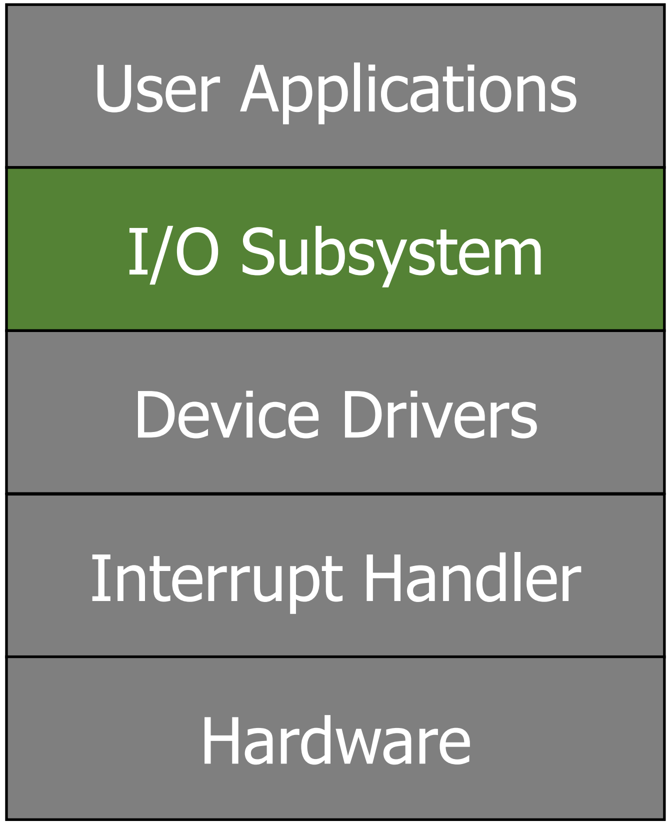


Outline

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

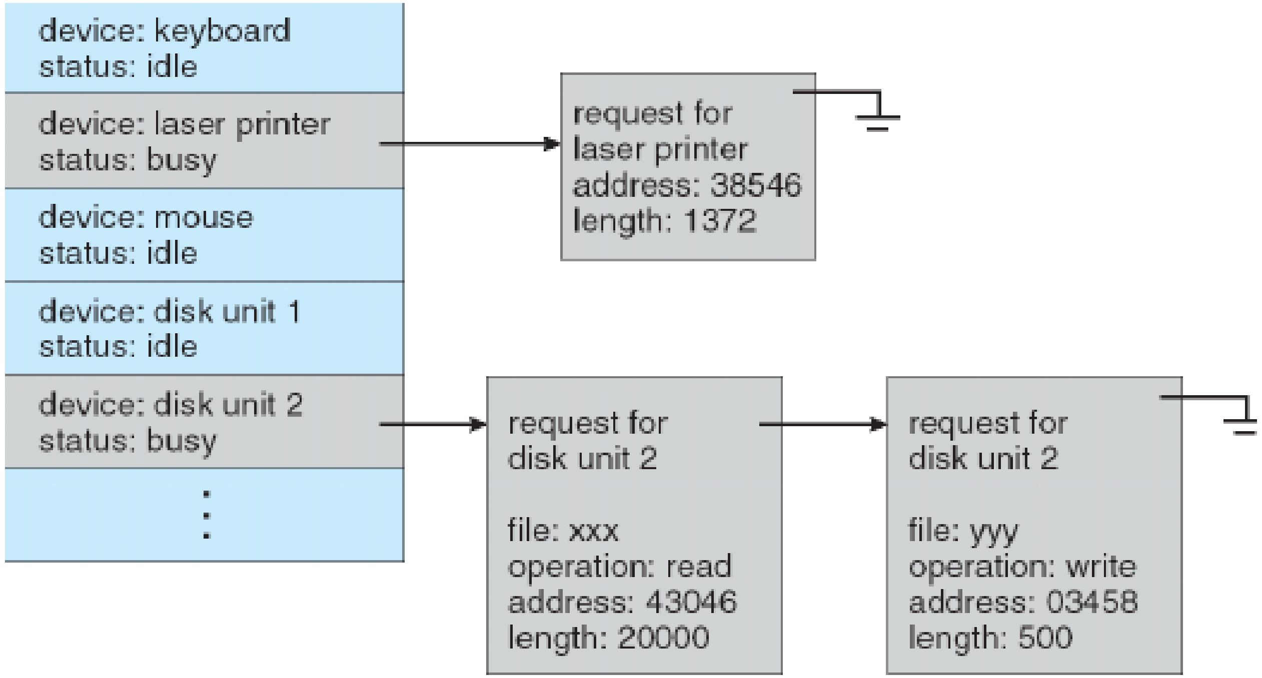
Where we are at in the system





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

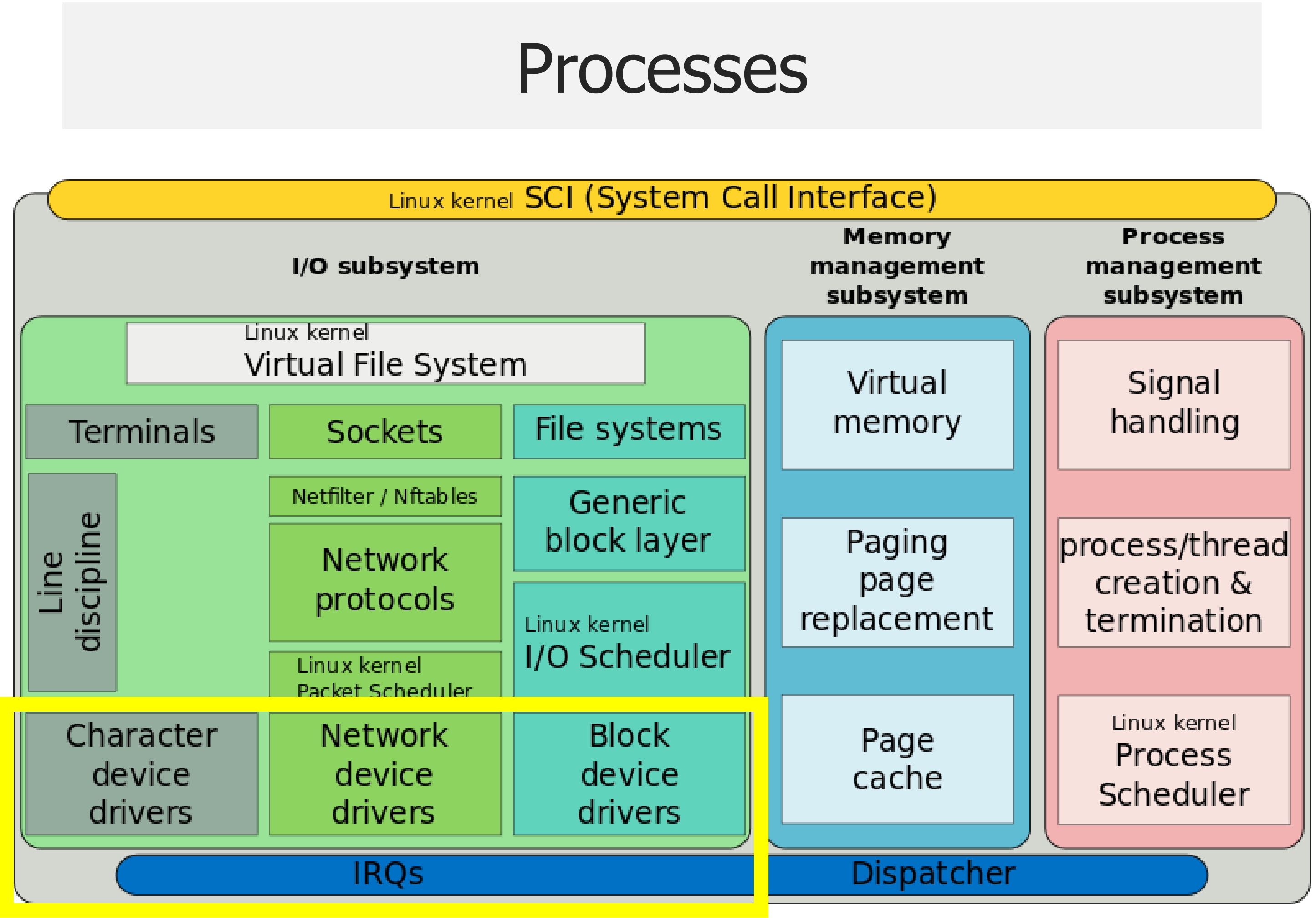
- Buffering
  - Kernel may need to hold on to a copy of data
    - Especially in asynchronous case
  - When copies are done and how many times is a big kernel efficiency question

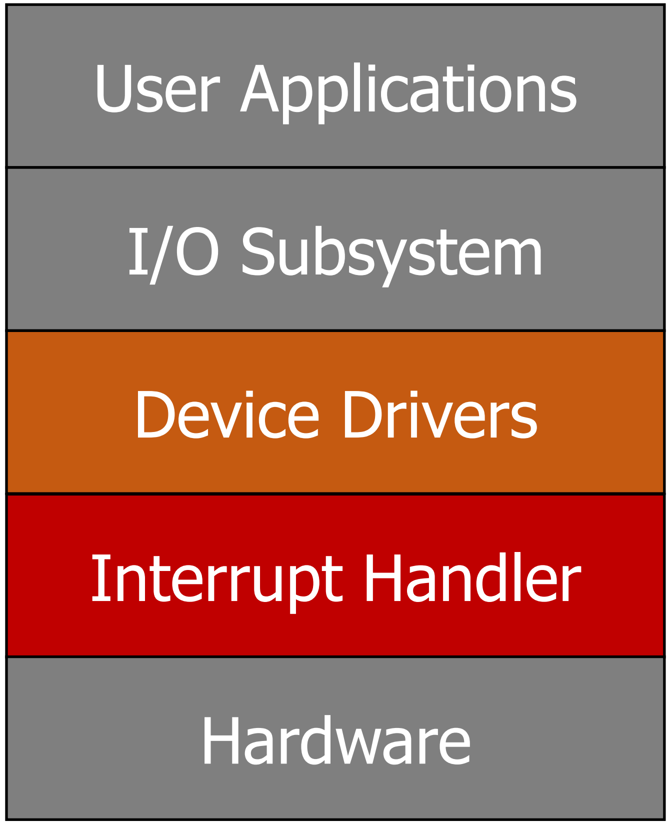
- Address translation
  - All the data user processes give to the kernel comes with virtual addresses
  - Pointers are either going to have to be translated
  - Or memory is going to need to be copied

Outline

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

Where we are at in the system

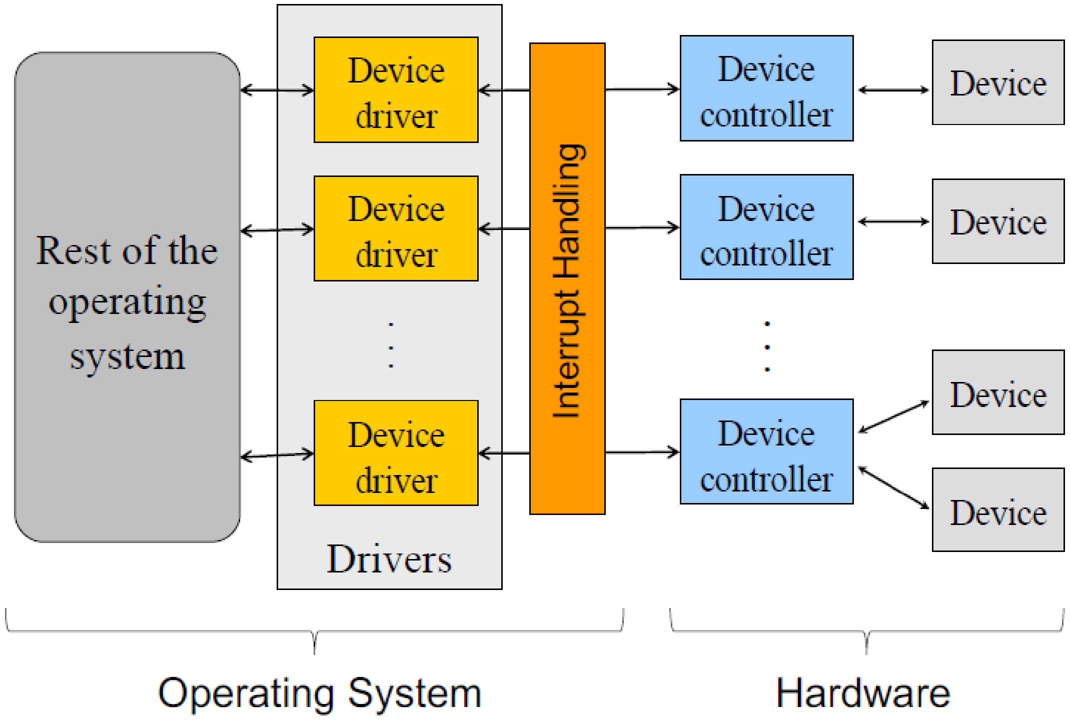




Device drivers

- 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

- Block device interface
- Various filesystems
- Generic SD Card Driver
- microSDHC UHS-I Driver
- Generic SPI Interface Driver
- Intel SPI Controller Driver
- Memory-Mapped I/O

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

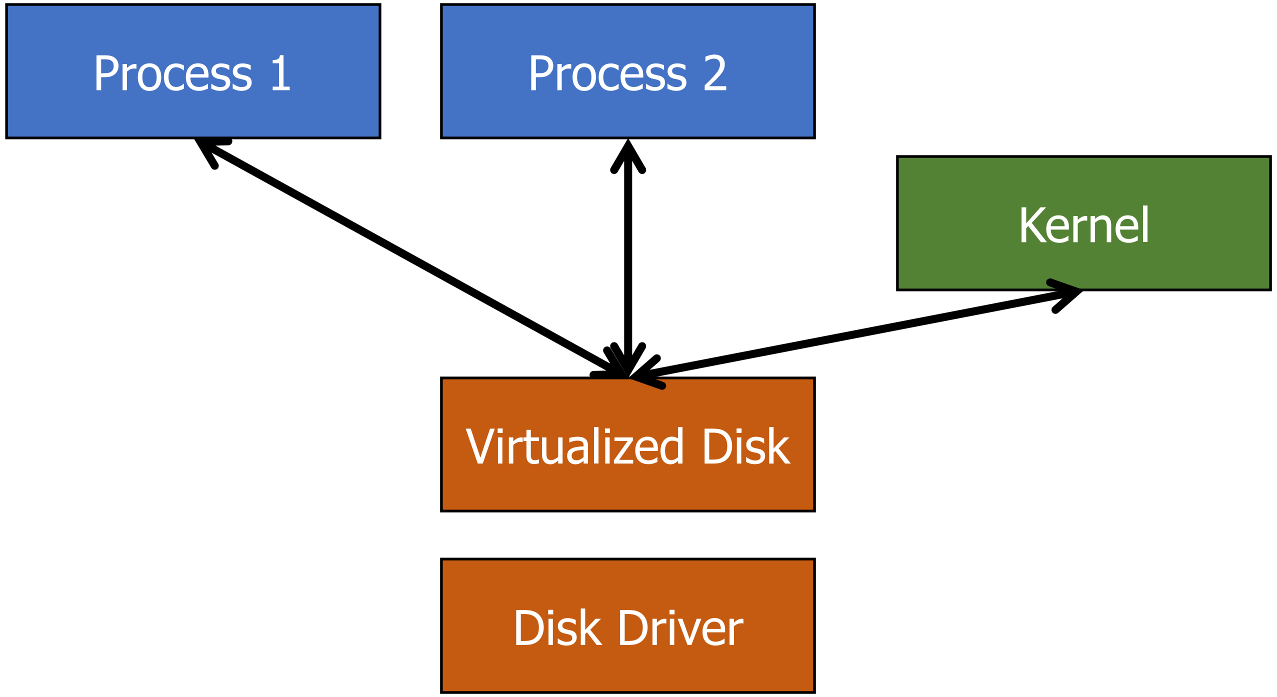
Device drivers are often designed with two “halves”

- Top half
  - 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

- Bottom half
  - Interrupt handler
    - Continues next transaction
    - Or signals for top half to continue (often with shared variable)

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

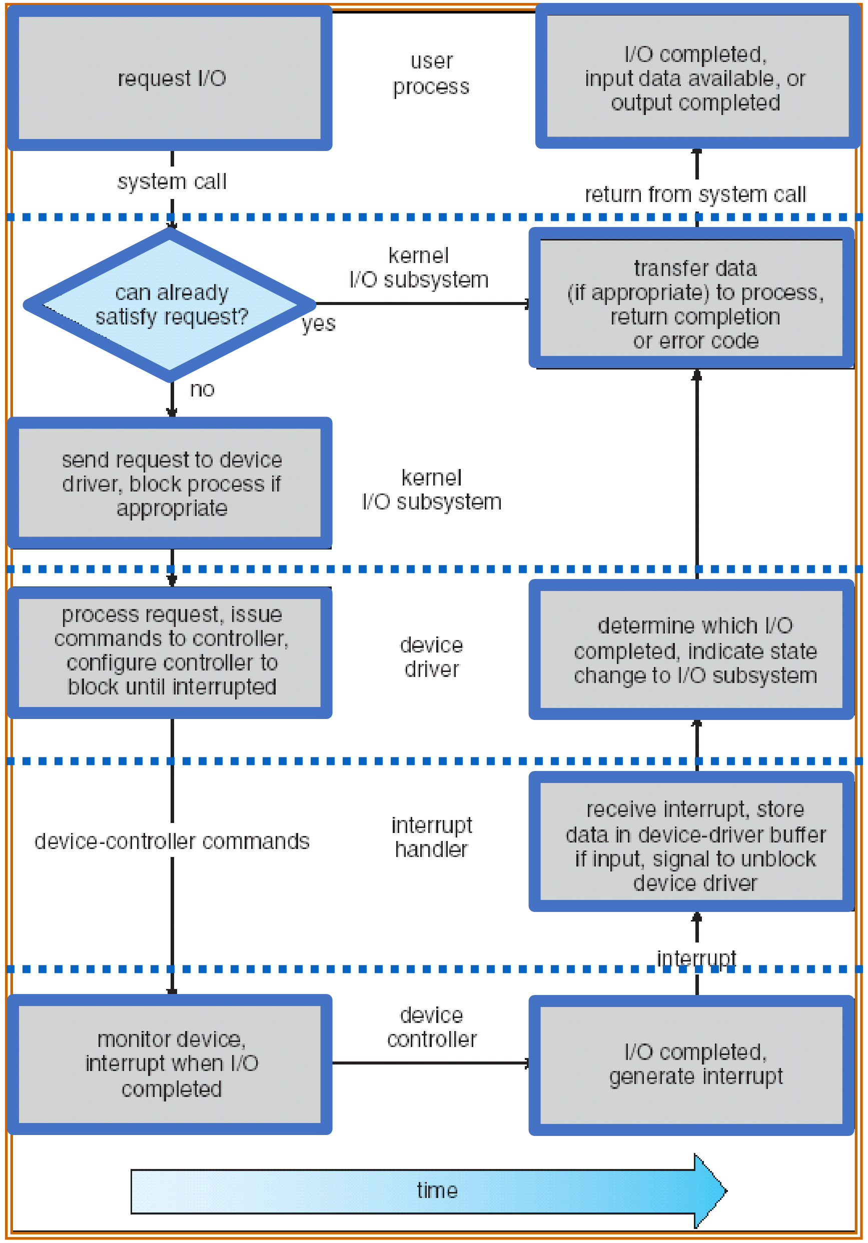
User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware



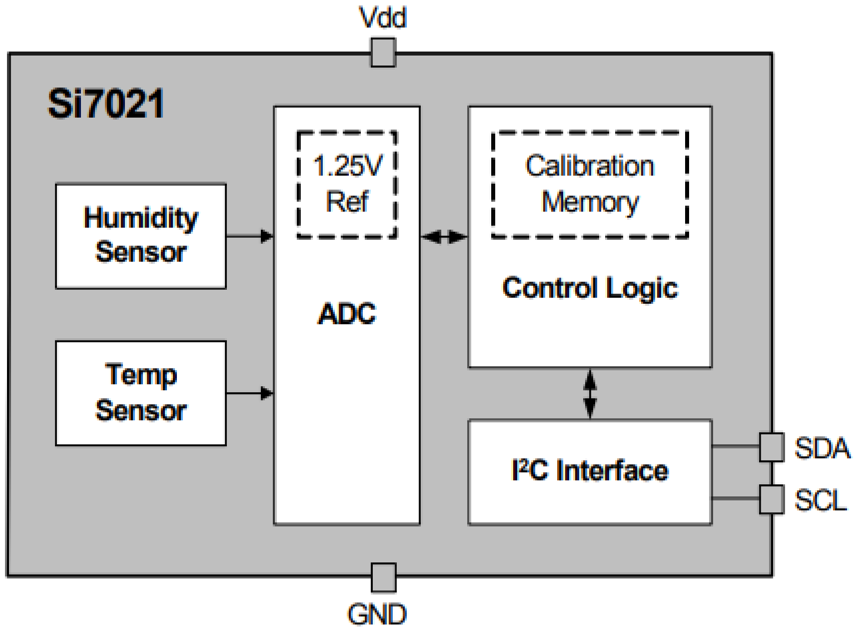
Outline

- Abstractions
- Device I/O layers
  - Application Layer
  - Kernel I/O Subsystem
  - Device Driver
  - Interrupt Handler
- **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 I^2C 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. I^2C Command Table

Command Description	Command Code
Measure Relative Humidity, Hold Master Mode	0xE5
Measure Relative Humidity, No Hold Master Mode	0xF5
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 I^2C controller (function calls)

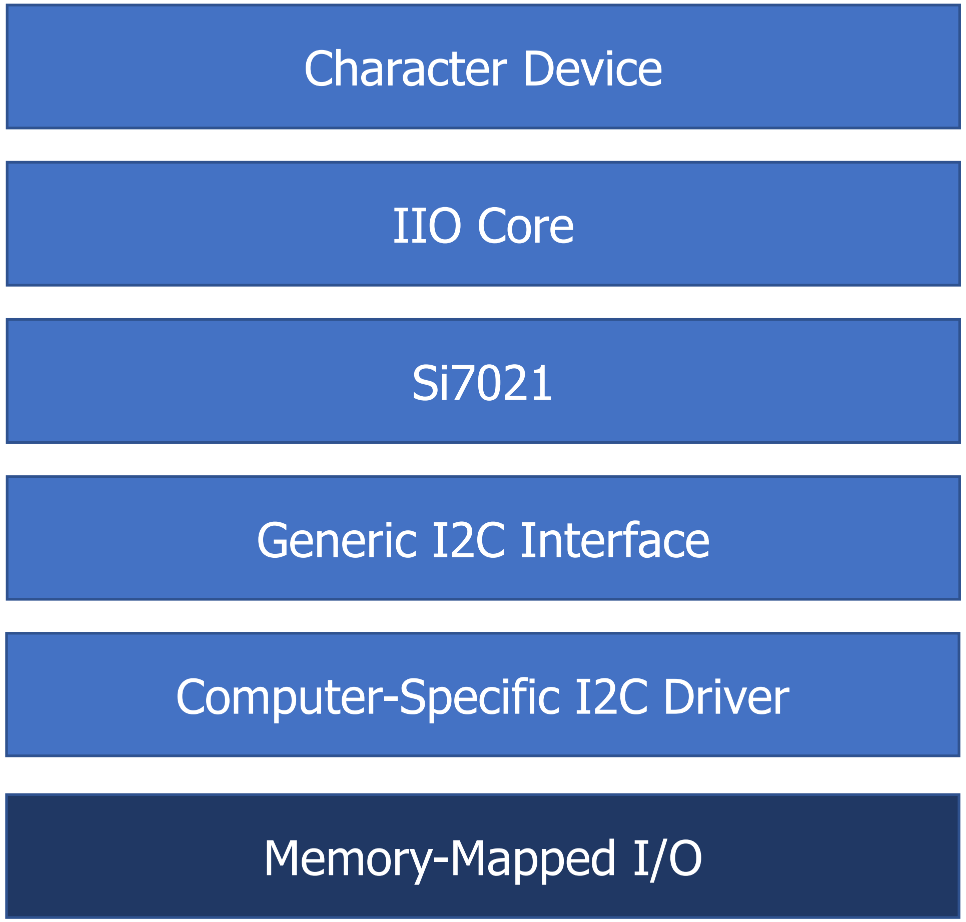
- In the driver we need to
  - See what the request from the layer above is
  - Perform an I^2C write transaction with a command byte (0xE3)
  - Wait until data is ready
  - Perform an I^2C read transaction to get the data
  - Translate the data into meaningful units

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

What are the driver layers going to be?

- In Linux, some sensors are connected through the Industrial I/O subsystem (IIO)
  - Handles sensor data specifically
    - Get raw sample
    - Get scaling value
    - Get offset value

- Lower layers could change and everything would still work
  - USB->I2C converter for example



Demo: Linux device driver code for Si7021

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

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 each device

- 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/src/si7021.rs

Outline

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