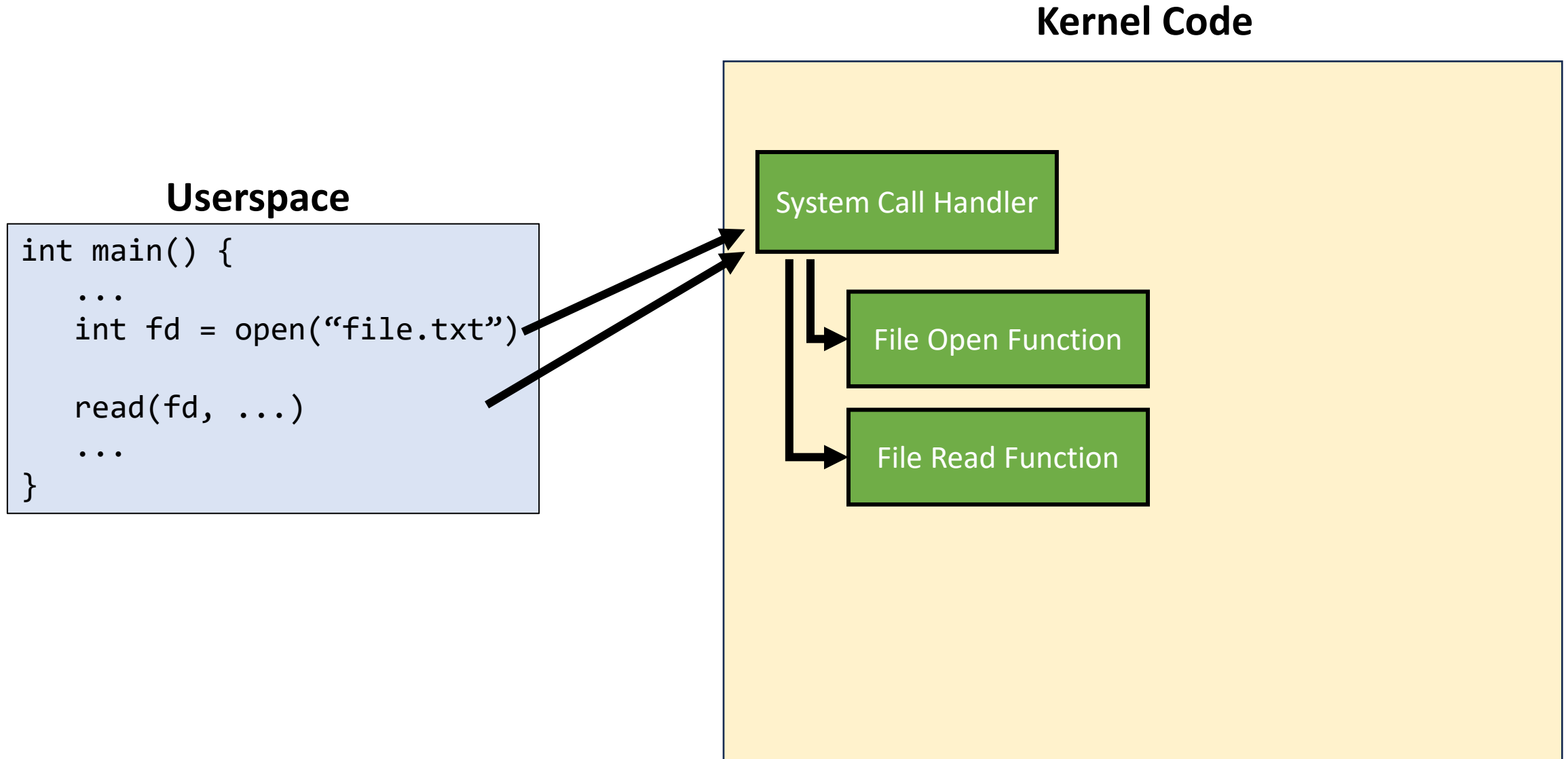


Devices

ECEN 427
Jeff Goeders

BYU Electrical & Computer
Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

Linux System Calls



At the end of last class we looked at the list of syscalls:

- <https://chromium.googlesource.com/chromiumos/docs/+/HEAD/constants/syscalls.md>

Where are the system calls to talk to hardware devices?

- UART
- Disk
- Video card
- Physical memory
- USB devices

Userspace

```
int main() {  
    ...  
    uart_get_input() ???  
    ...  
}
```

Don't our device drivers provide functions that we can call to access the devices?

- Look like ordinary files in the filesystem, but are special files
- They are interfaces to the driver that controls the device
- By using system calls that operate on files, our code can interface with the driver.
- They are not real files...

Have you used a device file before?

Pseudo-devices [\[edit\]](#)

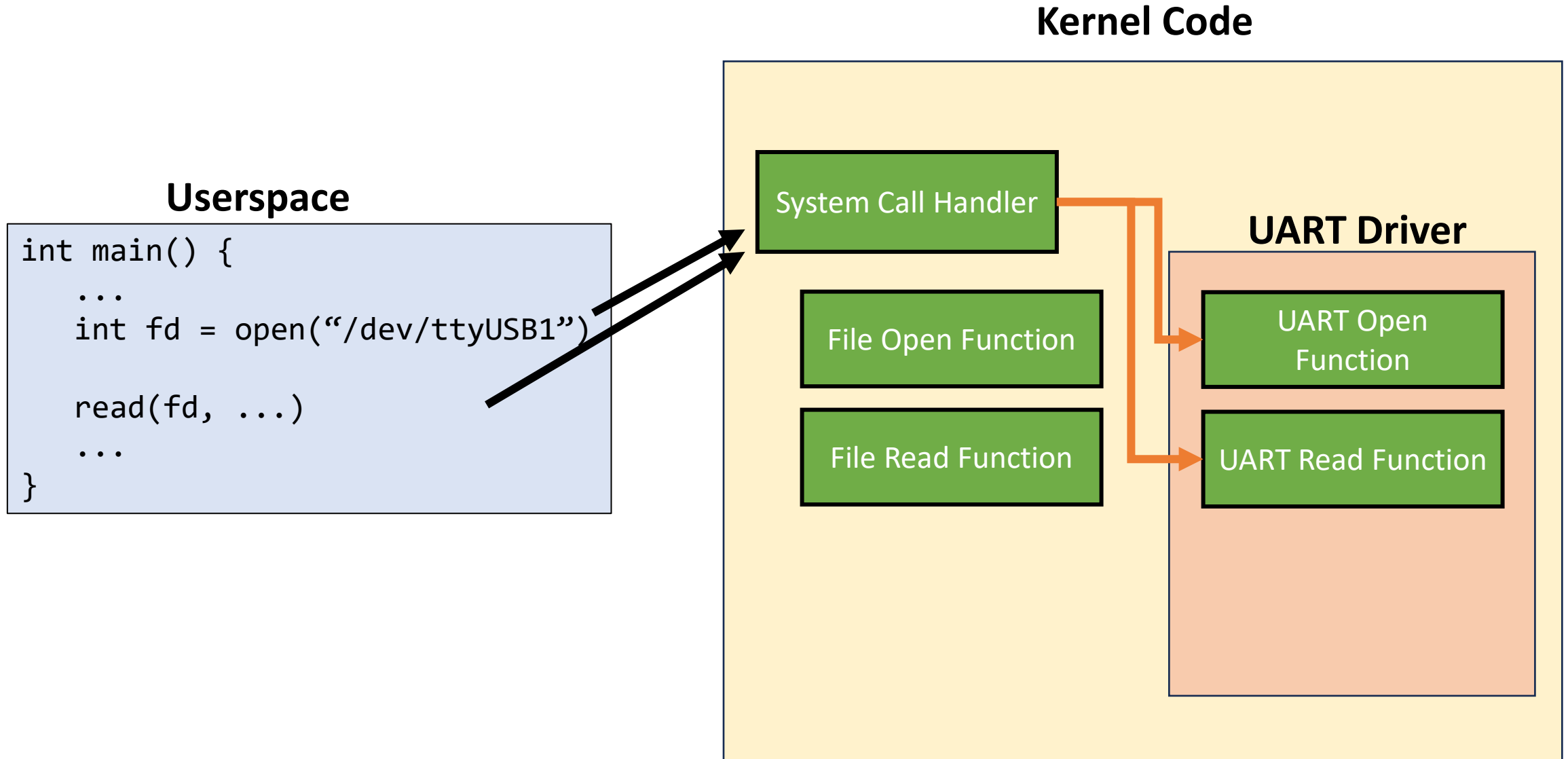
Device nodes on Unix-like systems do not necessarily have to correspond to [physical devices](#). Nodes that lack this correspondence form the group of *pseudo-devices*. They provide various functions handled by the operating system. Some of the most commonly used (character-based) pseudo-devices include:

- [/dev/null](#) – accepts and discards all input written to it; provides an [end-of-file](#) indication when read from.
- [/dev/zero](#) – accepts and discards all input written to it; produces a continuous stream of [null characters](#) (zero-value bytes) as output when read from.
- [/dev/full](#) – produces a continuous stream of null characters (zero-value bytes) as output when read from, and generates an [ENOSPC](#) ("disk full") error when attempting to write to it.
- [/dev/random](#) – produces bytes generated by the kernel's [cryptographically secure pseudorandom number generator](#). Its exact behavior varies by implementation, and sometimes variants such as [/dev/urandom](#) or [/dev/arandom](#) are also provided.
- [/dev/stdin](#), [/dev/stdout](#), [/dev/stderr](#) – access the process's [standard streams](#).
- [/dev/fd/n](#) – accesses the process's [file descriptor](#) *n*.

Additionally, BSD-specific pseudo-devices with an [ioctl](#) interface may also include:

- [/dev/pf](#) – allows userland processes to control [PF](#) through an [ioctl](#) interface.
- [/dev/bio](#) – provides [ioctl](#) access to devices otherwise not found as [/dev](#) nodes, used by [bioctl](#) to implement [RAID](#) management in [OpenBSD](#) and [NetBSD](#).
- [/dev/sysmon](#) – used by NetBSD's [envsys](#) framework for [hardware monitoring](#), accessed in the userland through [proplib\(3\)](#) by the [envstat](#) utility.^[8]

Linux System Calls



ECEN427 PYNQ Hardware System

- <https://byu-cpe.github.io/ecen427/documentation/hardware/>

Linux System Calls

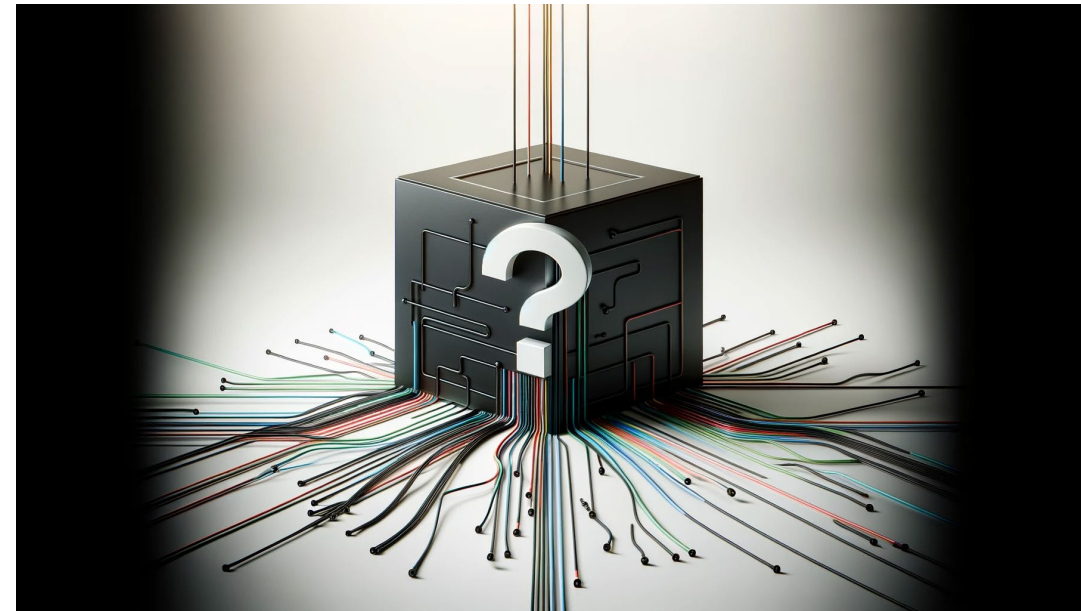
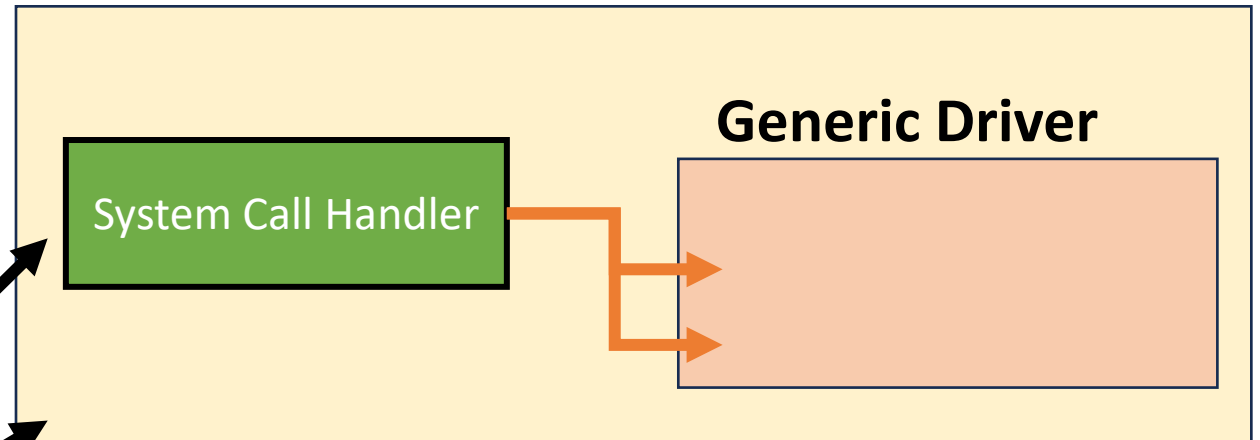
Do we really need a separate driver for every hardware device?

Can we make a generic driver?

Userspace

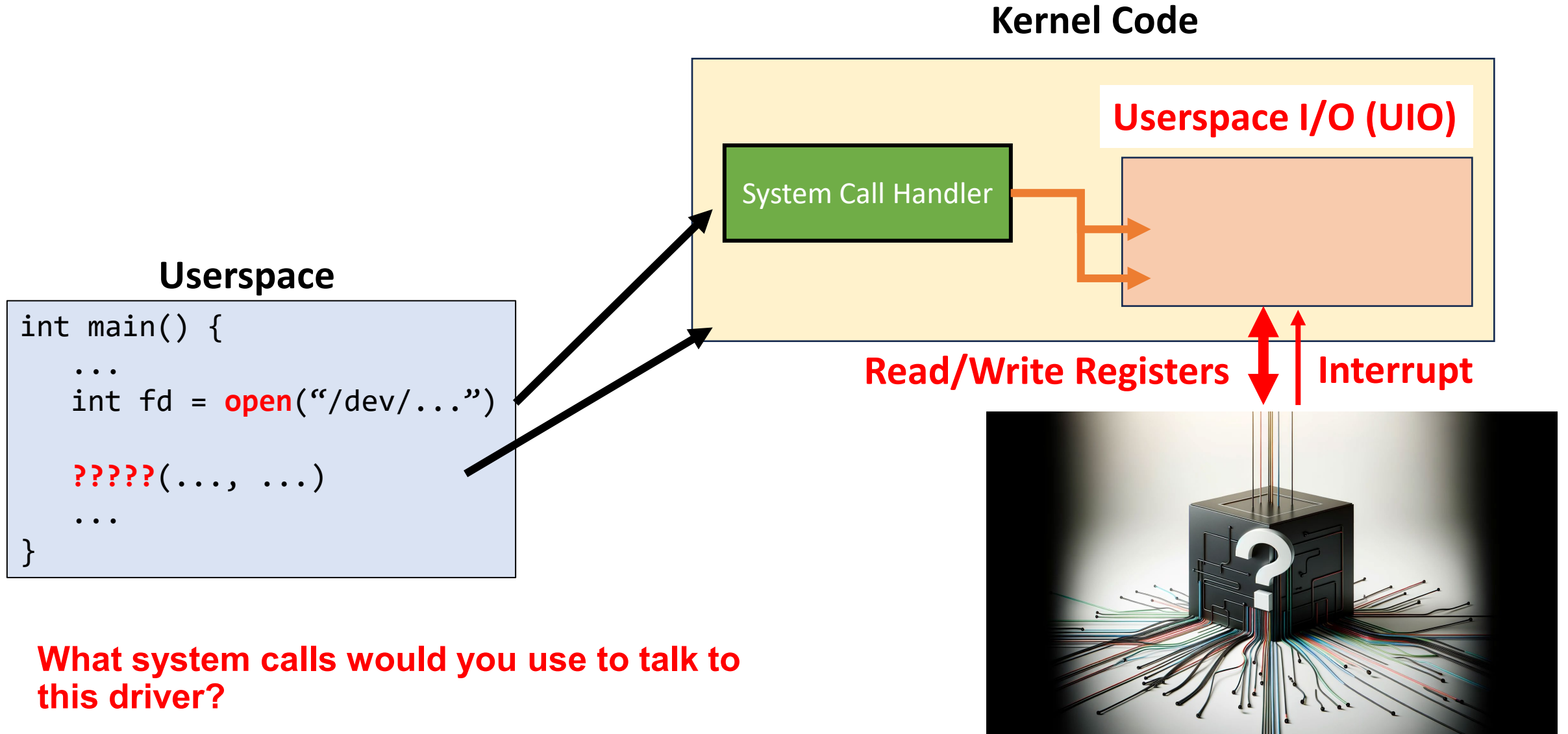
```
int main() {  
    ...  
    int fp = open("/dev/...")  
    ?????(..., ...)  
    ...  
}
```

Kernel Code



What kinds of things do you need
to do with a hardware device you
know nothing about?

Linux System Calls



What system calls would you use to talk to this driver?

Before learning more, let's pause and
cover a few more operating system
concepts...

`mmap()`

Map files (or devices) into memory

Process States

- Last lecture we talked briefly about processes, and how the operating system virtualizes the CPU in order to run multiple processes.
- The OS scheduler is responsible for selecting which process will run in the next **time slice**.
- Sometimes processes “block” and need to wait for an OS operation to complete.
 - Example: process calls `read()` to get data from a file on disk. This can take several milliseconds.
 - Rather than blocking all execution, the process goes to sleep, and the OS can wake it back up when the data is ready.

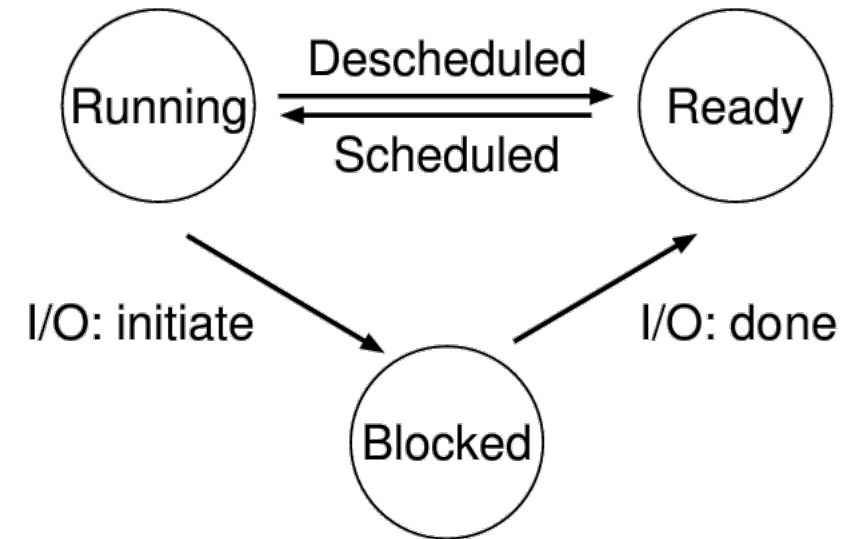


Figure 4.2: **Process: State Transitions**

The Linux Userspace I/O (UIO) driver, provides **generic** access to hardware devices allowing userspace to:

1. Access device memory addresses (read/write regs)
2. Check if device generated an interrupt

- **Read/Write addresses (registers):**
 - `mmap()`
 - Access registers via pointer returned from `mmap`
- **Enable interrupts:**
 - `write()` a value of 1
 - *You must write 4 bytes*
- **Block/wait for interrupt :**
 - `read()`
 - *You must read 4 bytes*
- **Check for interrupt (non-blocking):**
 - `poll()`
 - Used to check if file has data to read

ECEN427 PYNQ Software System

- <https://byu-cpe.github.io/ecen427/documentation/software-stack/>

- The UIO linux driver is not meant to actually serve as the device driver
- It provides an interface that allows you to write drivers in userspace.
- In lab 2, you will create userspace drivers for the buttons, switches and interrupt controller
 - These will interface with the UIO kernel driver to access the devices
- Example https://github.com/byu-cpe/ecen427_student/blob/main/userspace/drivers/uio_example/generic_uio_example.c