Operating System Design and Implementation

Lecture 14: I/O memory

Tsung Tai Yeh

Tuesday: 3:30 – 5:20 pm
Classroom: ED-302

Acknowledgements and Disclaimer

- Slides was developed in the reference with
  MIT 6.828 Operating system engineering class, 2018
  MIT 6.004 Operating system, 2018
  Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC
  Onur Mutlu, Computer architecture, ece 447, Carnegie Mellon University

Outline

- I/O hardware
- Memory-mapped I/O
- Direct memory access (DMA)

I/O hardware

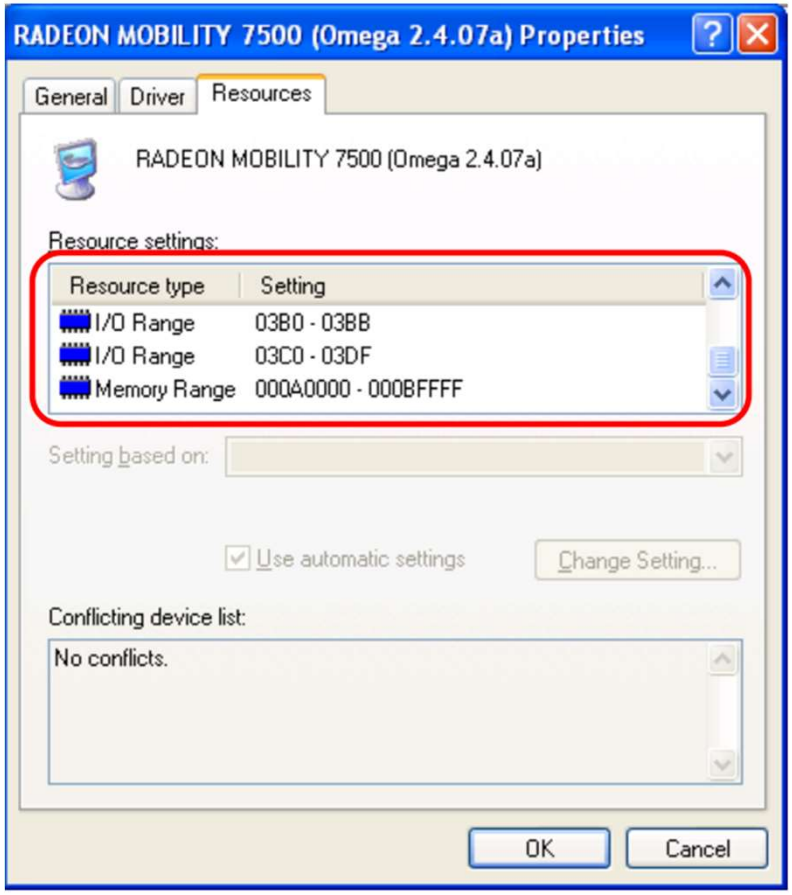
- The variety of I/O devices
  - Storage, communication
- Common concepts for I/O hardware
  - **Bus**: an interconnection between components
  - **Port**: connection point for device
  - **Controller**: component that controls the device
    - Can be integrated to device or separate circuit board
    - Usually contains processor, microcode, private memory, bus controller, etc..
  - I/O access can use **polling** or **interrupt**

I/O hardware

- Some CPU architecture has dedicated I/O instructions
  - E.g. x86: in, out, ins, outs
- Devices usually provide registers for data and control I/O
  - Device driver places commands and data to register
  - **Data**(in/out), **status, control** (command) register
  - Typically 1 – 4 bytes, or FIFO buffer
- Devices are assigned addresses for registers or on-device memory
  - **Direct I/O instructions**
  - **Memory-mapped I/O**

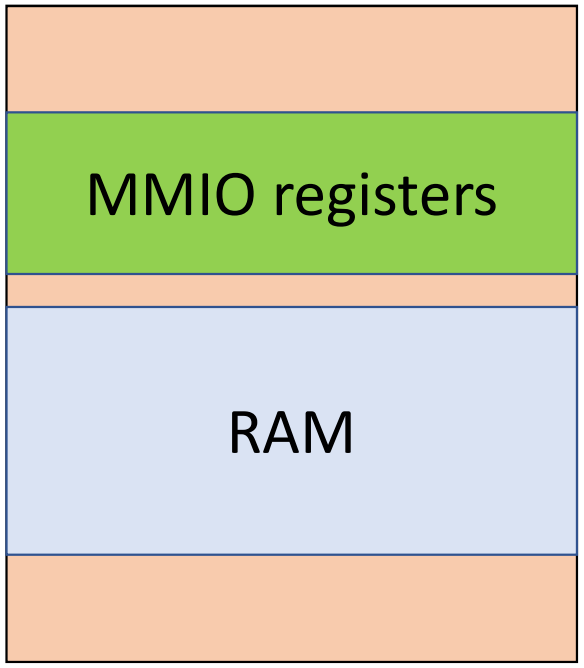
Communicating with devices

- Most devices can be considered as memories
  - With an “address” for R/W
  - To transfer data to or from a particular device, the CPU can access special addresses
  - Here, a video card can be accessed via addresses 3B0-3BB, 3C0 – 3DF and A0000 – BFFFF
  - There are two ways these addresses can be accessed



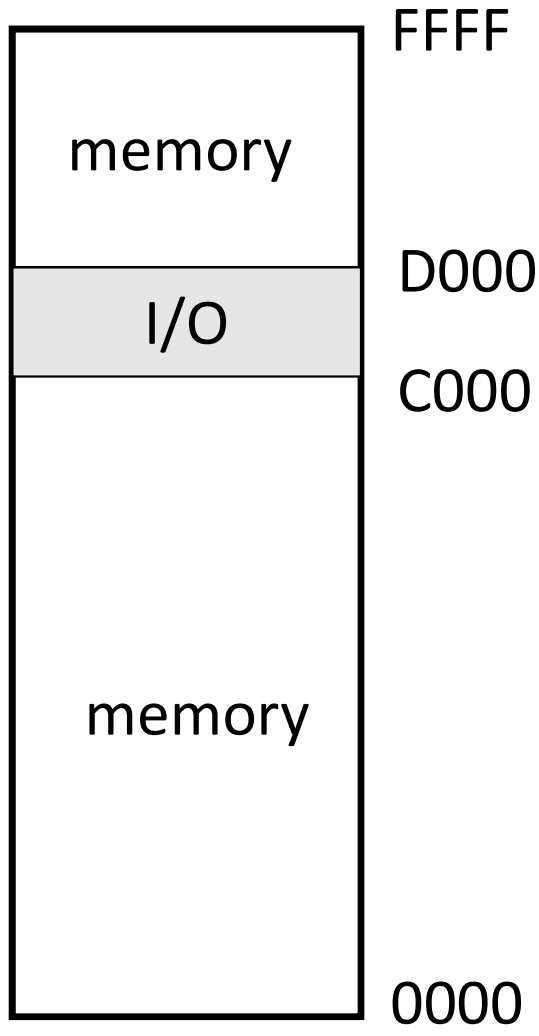
Memory-mapped I/O

- **Memory-mapped I/O**
  - Data and command registers **mapped to processor address space**
  - Access to the **I/O device registers using normal load/store instruction**
  - Most widely used I/O method across the different architecture supported by Linux

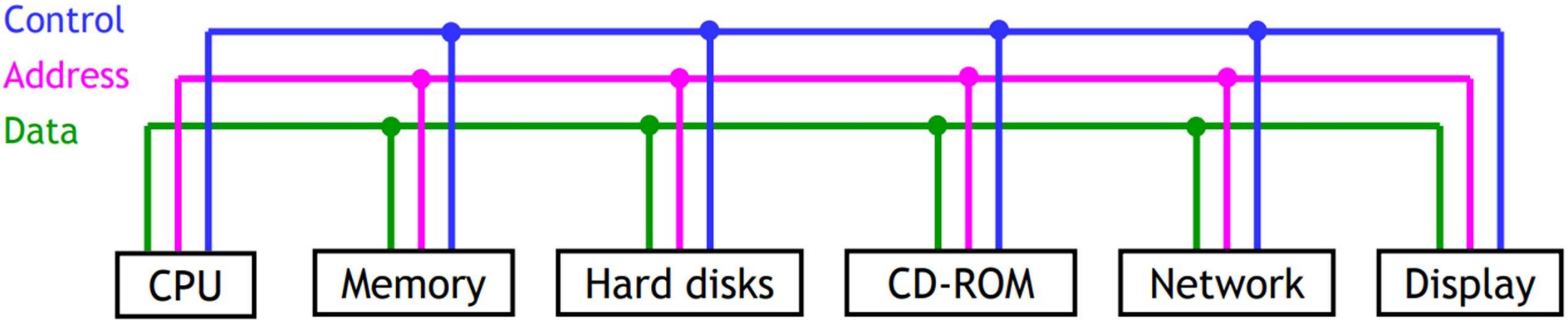


Memory-mapped I/O

- With *memory-mapped I/O*
  - One address space is divided into two parts
  - Addresses for I/O devices
  - Other addresses did reference main memory
  - I/O addresses are shared by many peripherals
    - E.g. Apple IIe, C010 is attached to the keyboard where C030 goes to the speaker



Programming memory-mapped I/O

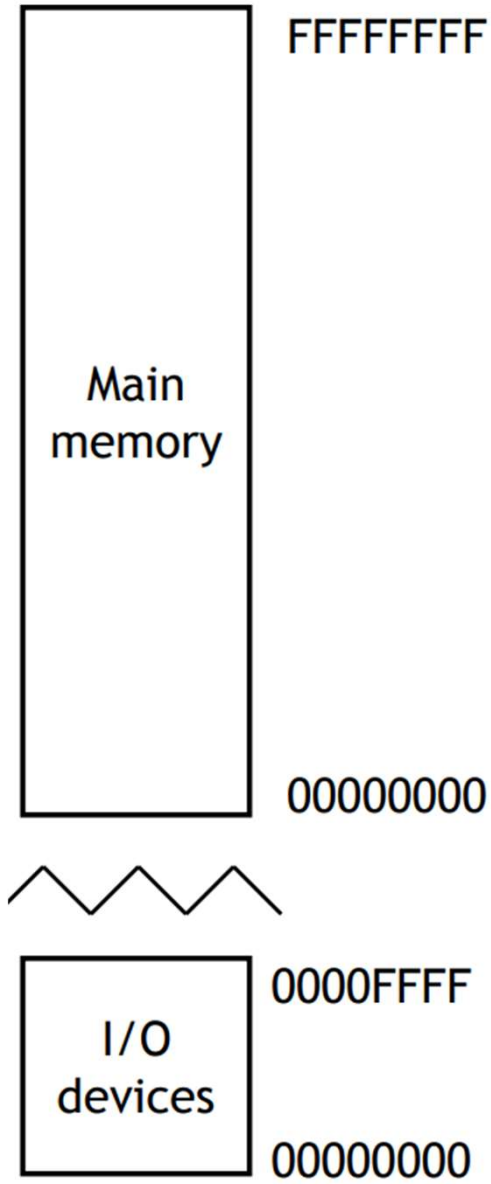


- The CPU sends data to appropriate I/O address
  - The address and data are also transmitted along the bus
- Each device monitors the address bus to see if it is the target
  - The speaker only responds when C030 appears on the address bus

https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec21.pdf

Isolated I/O

- Isolated I/O
  - Separate address spaces for memory and I/O devices
  - With special instructions that access the I/O space
- In 32-bit address space, 8086 machines
  - Regular instructions like **MOV** reference RAM
  - The special instructions **IN** and **OUT** access a separate 64 KB I/O address space



https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec21.pdf

Memory-mapped v.s. isolated I/O

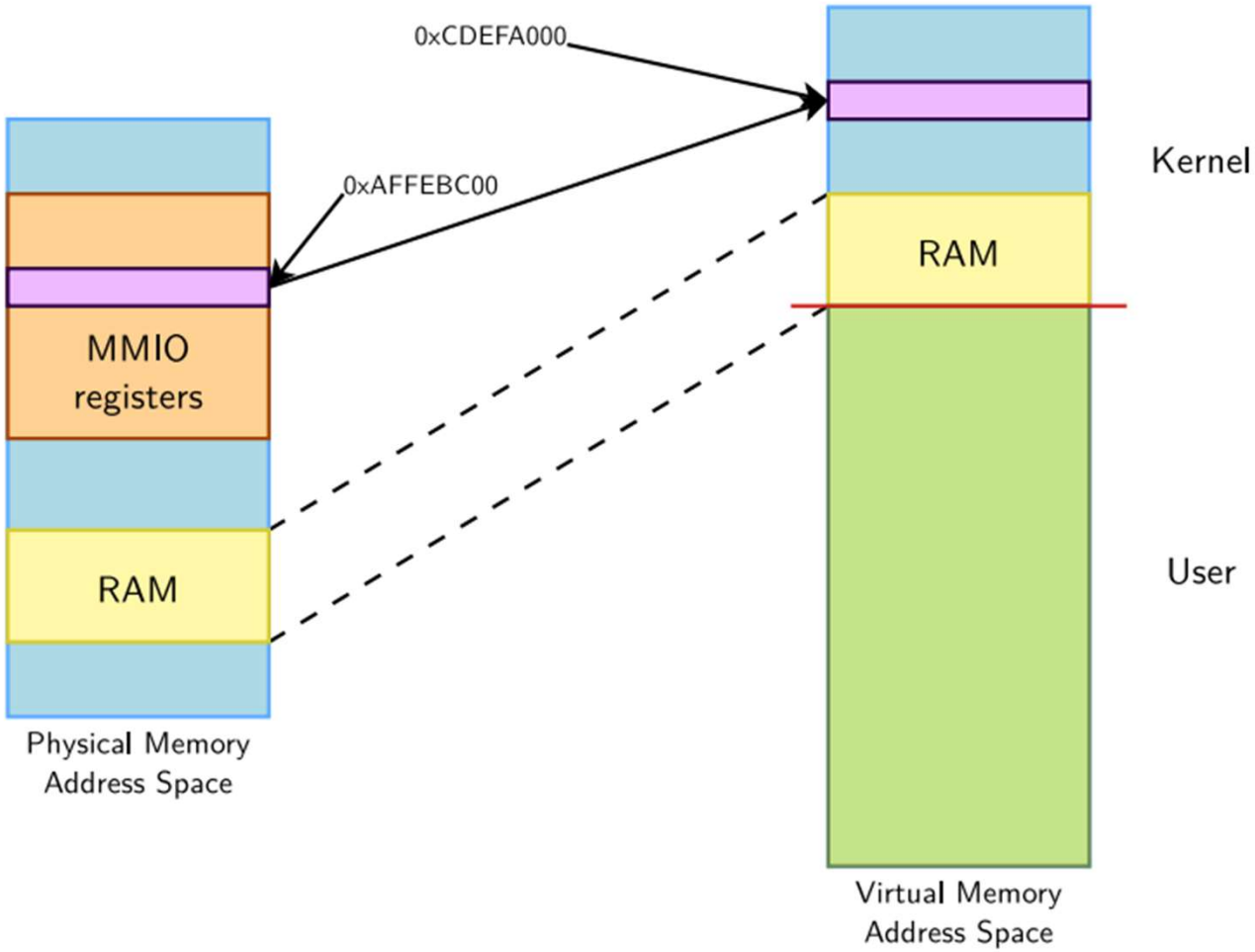
- Memory-mapped I/O
  - A single address space is nice
  - The same instructions that access memory can also access I/O devices
  - Issuing MIPS sw instructions to the proper addresses can store data to an external device
- Isolated I/O
  - Special instructions are used to access devices
  - This is less flexible for programming

Mapping I/O memory

- Load/store instructions work with virtual addresses
- To access I/O memory
  - Drivers need to have a virtual address that the processor can handle
  - I/O memory is not mapped by default in virtual memory
- The **ioremap** function satisfies this need:

```
#include <asm/io.h>
void __iomem *ioremap(phys_addr_t phys_addr, unsigned long size);
void iounmap (void __iomem *addr);
```

ioremap()




```
ioremap(0xAFFEBC00, 4096) = 0xCDEFA000
```

https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Managed API

- request_mem_region() and ioremap() is now deprecated
- Using below managed functions instead
  - devm_ioremap(), devm_iounmap()
  - devm_ioremap_resource()
    - Takes care of both the request and remapping operations

Access MMIO devices

- To do PCI-style, little-endian access
  - unsigned read[bwlq](void *addr);
  - void write[bwlq] (unsigned val, void *addr);
- To do raw access, without endianness conversion
  - unsigned __raw_read[bwlq] (void *addr);
  - void __raw_write[bwlq] (unsigned val, void *addr);
  - For example:

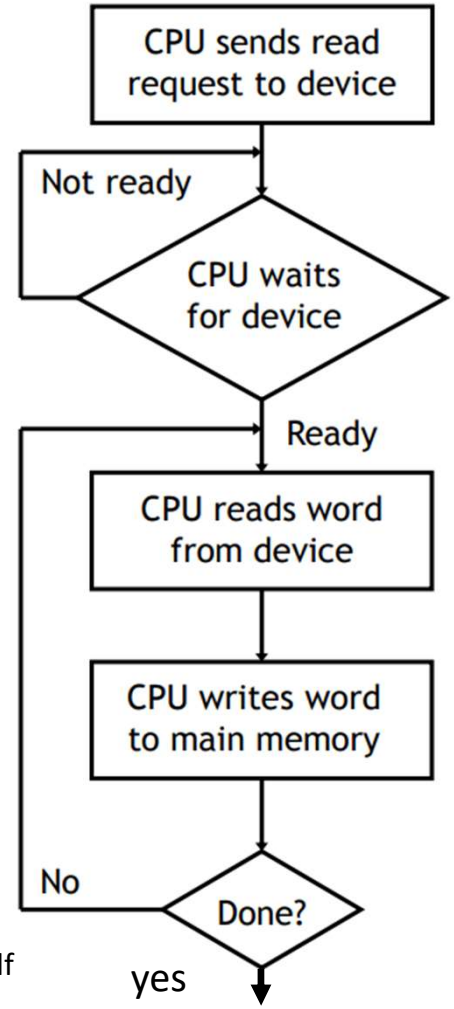
```
32 bit write (drivers/tty/serial/uartlite.c)
writel(c & 0xff, port->membase + 4);
```

Avoid I/O access issues

- The compiler and/or CPU can reorder memory accesses
  - Might cause trouble for devices is they expect one register to be read/written before another one
  - **Memory barriers are available** to prevent this reordering
  - rmb() is a read memory barrier, prevents reads to cross the barrier
  - wmb() is a write memory barrier
  - mb() is a read-write memory barrier
  - Note that readl(), writel() and similar functions already contain barriers

How data between a device and memory ?

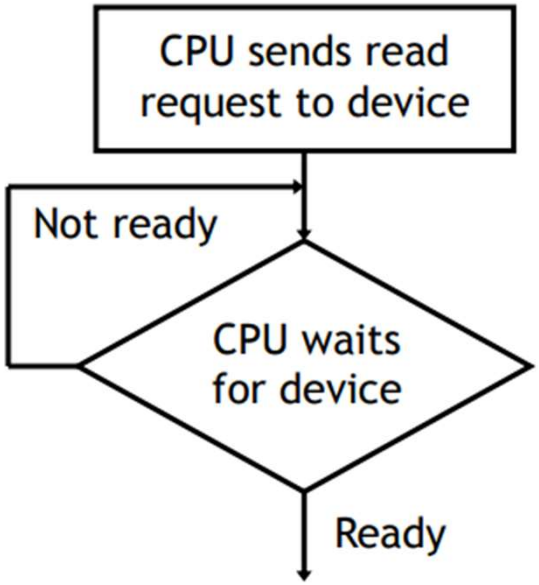
- **Programmed I/O**
  - The CPU makes a request and waits for the device to be ready
  - Buses are only 32-64 bits wide
  - How to do for large data transfers ?
    - Repeated writes words to main memory
  - A lot of CPU time is needed for this !!
    - If the device is slow the CPU might have to wait for a long time
    - The CPU is involved as a middleman for the actual data transfer



https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec21.pdf

Polling

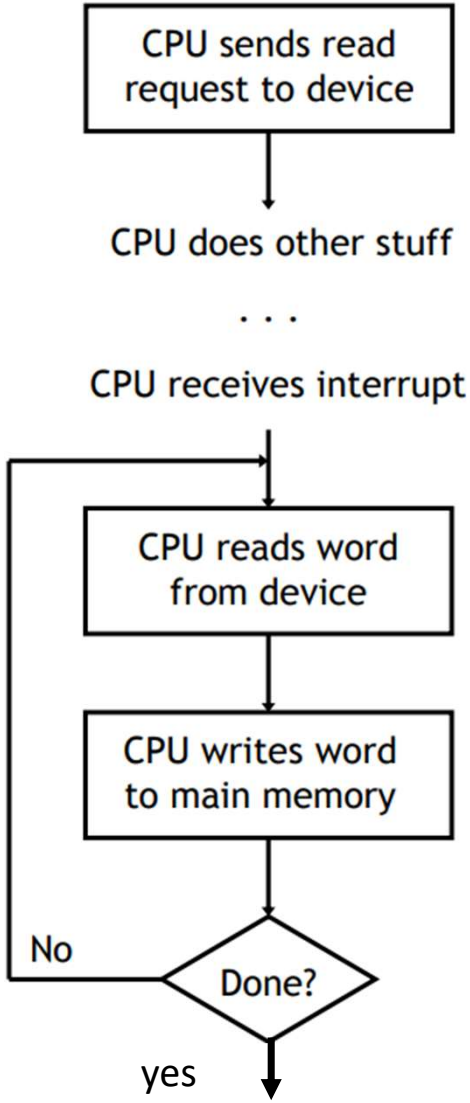
- **Polling**
  - Continually checking to see if a device is ready
  - It is not an efficient use of the CPU
  - Most devices are slow compared to modern CPUs
  - The processor has to ask often enough to ensure that it doesn't miss anything
  - The CPU cannot do much else while waiting



https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec21.pdf

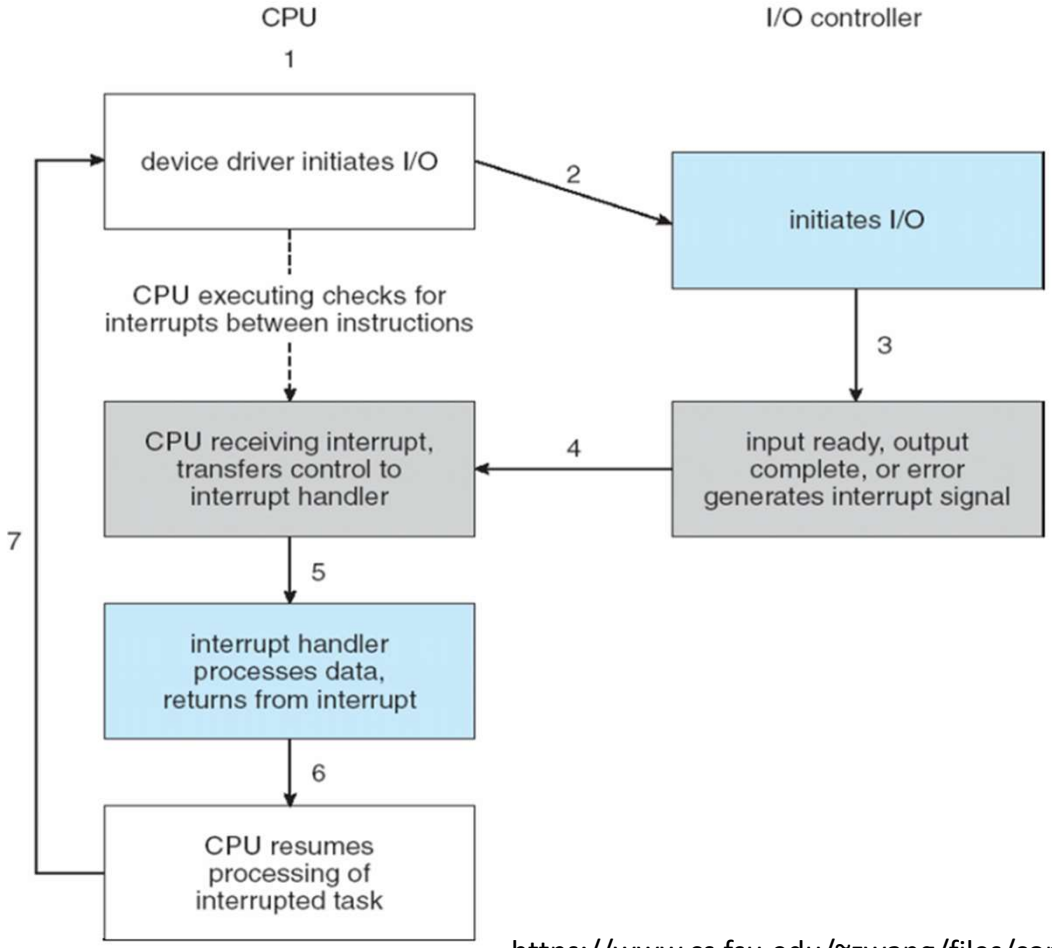
Interrupt-driven I/O

- **Interrupt-driven I/O**
  - **The device interrupts the processor** when the data is ready
  - The data transfer steps are the same as with programmed I/O, and still occupy the CPU



https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec21.pdf

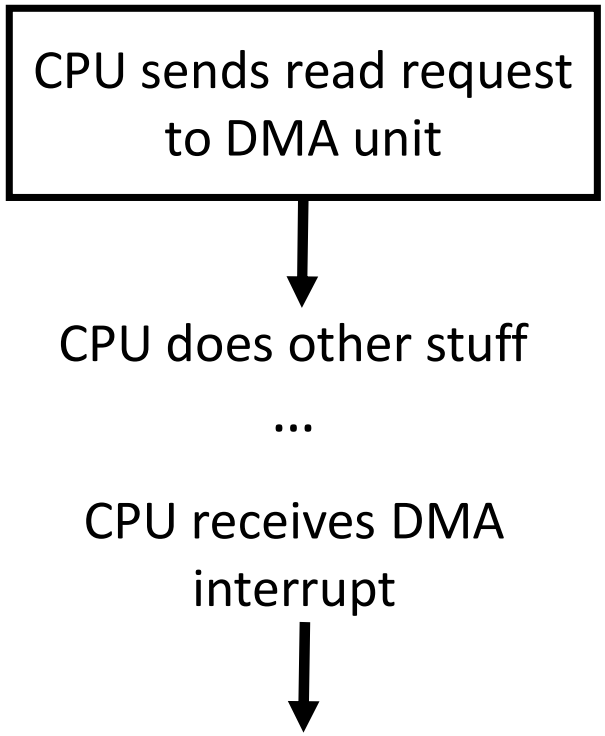
Interrupt-driven I/O steps



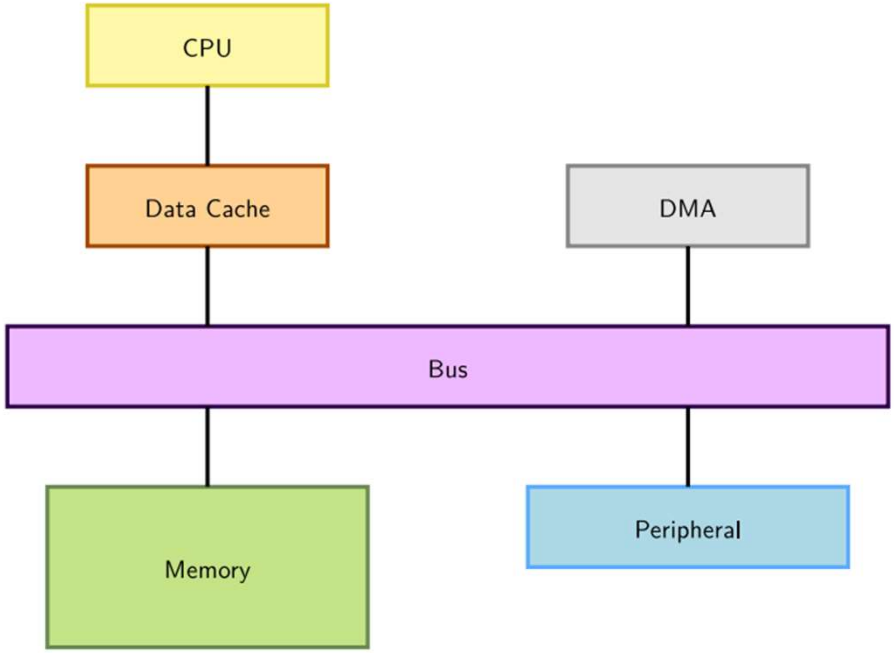
https://www.cs.fsu.edu/~zwang/files/cop4610/Fall2016/chapter13.pdf

Direct memory access (DMA)

- **Direct memory access (DMA)**
  - **Copy data directly between devices and RAM** and bypass the CPU
  - OS issues commands to the DMA controller
  - The pointer of the command written into the command register
  - When done, device interrupts CPU to signal completion



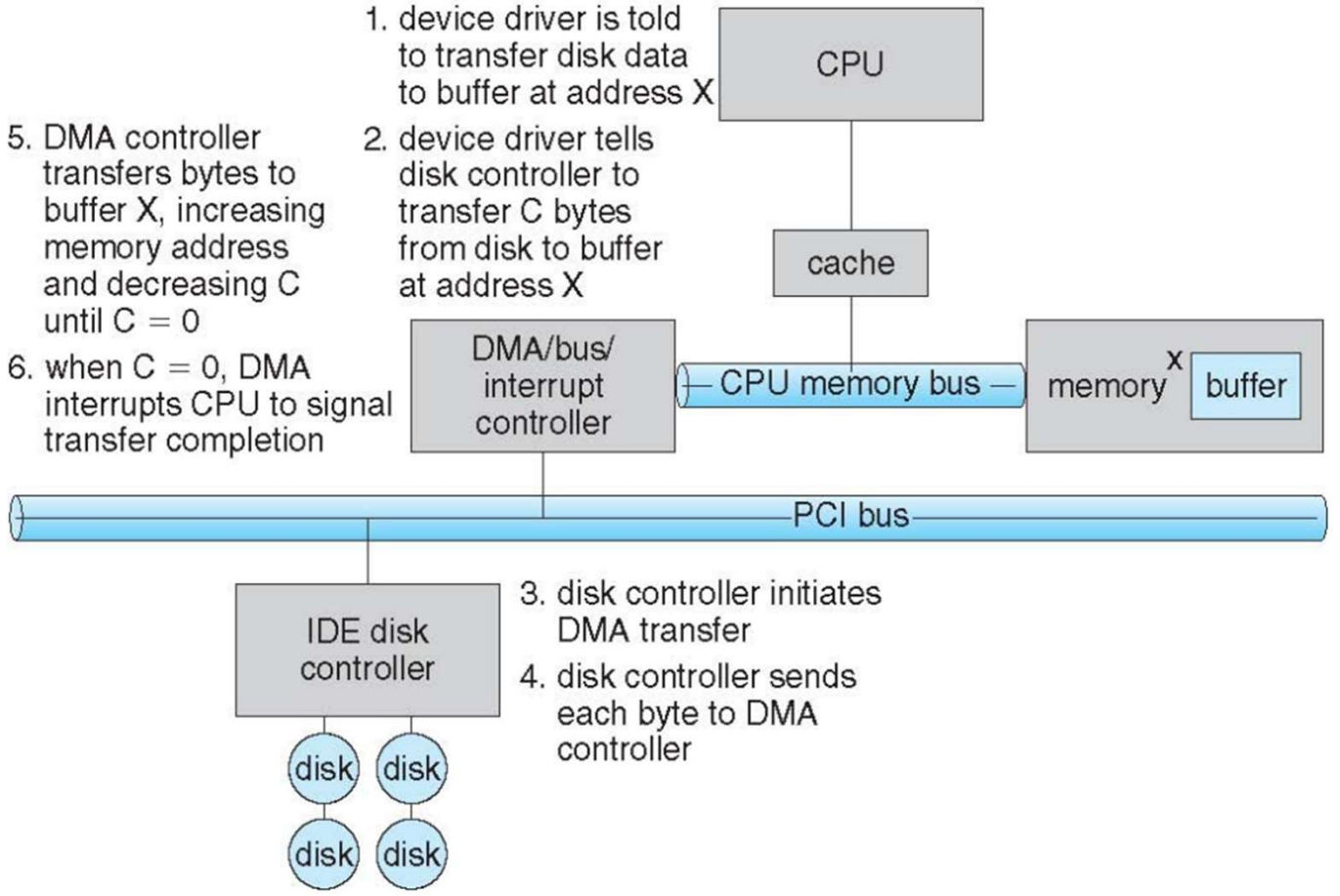
DMA controller



- The DMA controller is a simple processor
  - The CPU asks the DMA controller to transfer data between a device and main memory
  - After that, the CPU can continue with other tasks
  - The DMA controller issues requests to the right I/O device
  - DMA waits and manages the transfers between the device and main memory
  - Once completed, the DMA controller interrupts the CPU

https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

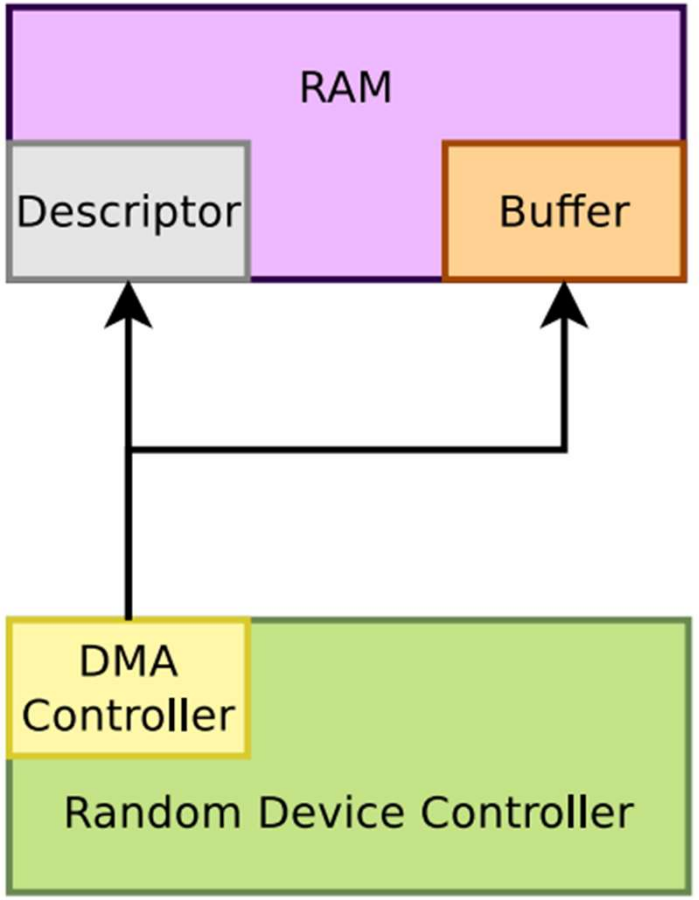
Six steps of DMA transfer



https://www.cs.fsu.edu/~zwang/files/cop4610/Fall2016/chapter13.pdf

Peripheral DMA

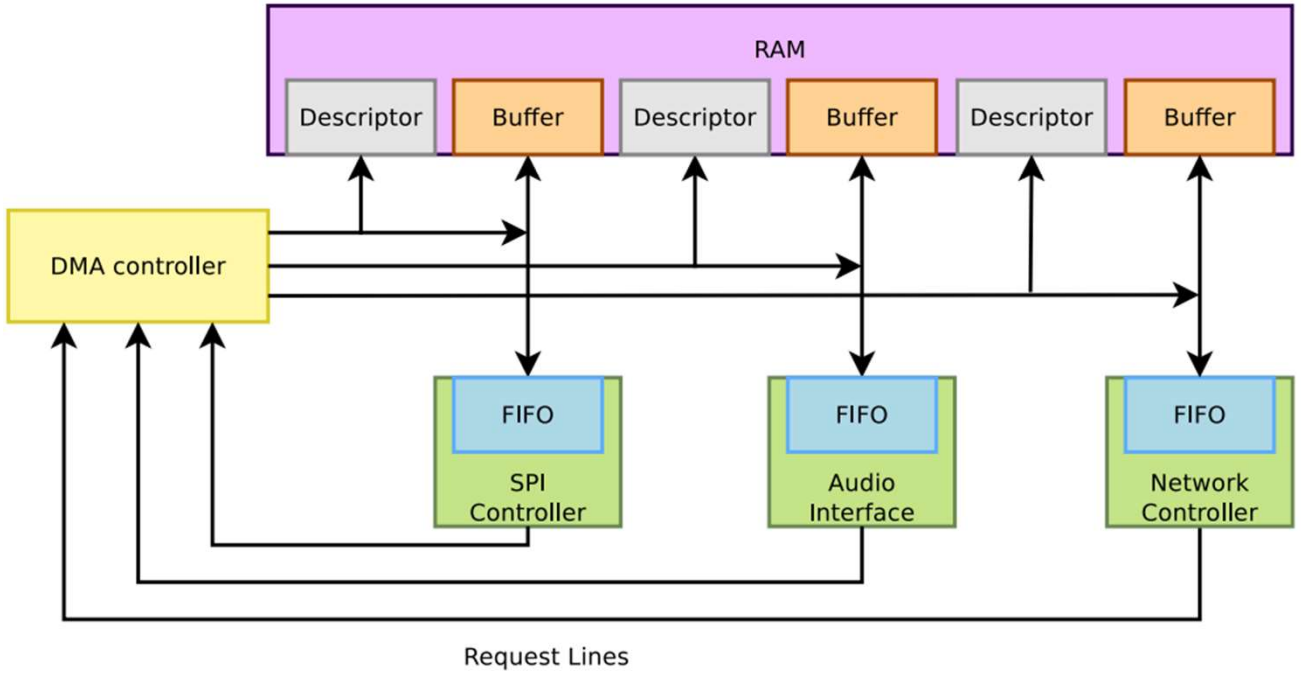
- Some device controller embedded their own DMA controller
  - Can do DMA on their own



https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

DMA controllers

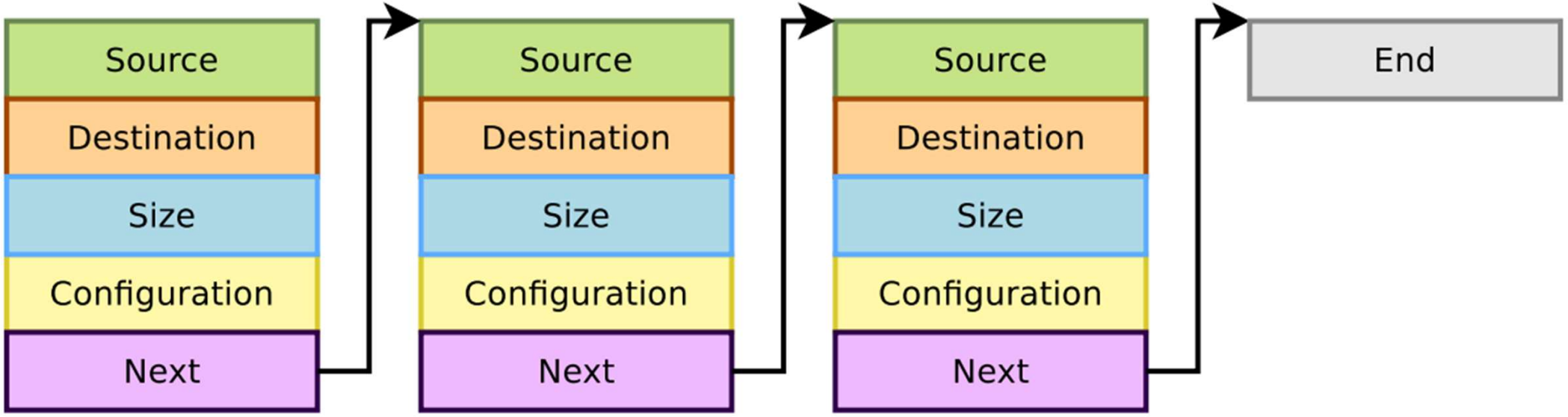
- An external DMA controller (on the SoC)
- Their drivers need to submit DMA descriptors to this controller



https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

DMA descriptors

- DMA descriptors describe the various attributes of a DMA transfer, and are chained



https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Constraints with a DMA

- A DMA deals with physical addresses
  - The memory accessed by the DMA shall be physically contiguous
- The CPU can access memory through a data cache
  - Using the cache can be more efficient (faster accesses to the cache than the bus)
  - The DMA does not access the CPU cache
  - Need to take care **cache coherency**
  - Either clean (write back to memory) or invalidate the cache lines corresponding to the buffer accessed by DMA and processor at the right times

DMA memory constraints

- Need to use contiguous physical memory space
- Can allocate memory by using
  - kmalloc() (up to 128 KB) or __get_free_pages() (up to 8 MB)
- Cannot use
  - vmalloc()

Memory synchronization issues

- Memory caching could interfere with DMA
  - Before DMA to device
    - Need to make sure all writes to the DMA buffer are completed
    - Corresponding cache lines are cleaned
  - After DMA from device
    - Before drivers read from a DMA buffer, need to ensure that the corresponding cache lines are invalidated

Linux DMA API

- The kernel DMA utilities can take care of
  - Either allocating a buffer in a cache coherent area
  - Or making sure caches are handled when required
  - Managing the DMA mappings and IOMMU
  - Most subsystems (such as PCI or USB) supply their own DMA API
  - See core-api/dma-api for details in the Linux DMA generic API

Coherent DMA mappings

- Coherent mappings
  - The kernel allocates a suitable buffer and sets the mapping for the driver
  - Can simultaneously be accessed by the CPU and device
  - Has to be in a cache coherent memory area
  - Usually allocated for the whole time the module is loaded

```
#include <asm/dma-mapping.h>
void *dma_alloc_coherent (struct device *dev, size_t size, dma_addr_t
*handle, gfp_t gfp);
void dma_free_coherent (struct device *dev, size_t size, void *cpu_addr,
dma_addr_t handle);
```

Starting DMA transfers

- In peripheral DMA
  - No external API is involved
- In external DMA controller
  - Ask the hardware to use DMA, so that it will drive its request line
  - Use Linux DMAEngine framework such as slave API

DMAEngine slave API

- DMA transfer with DMAEgnine by using following functions in the driver
  - Request a channel for exclusive use with dma_request_channel()
  - Configure it for our use case by filling a struct dma_slave_config and pass it as an argument to dmaengine_slave_config()
  - Start a new transaction with dmaengine_pre_slave_single() or dmaengine_pre_slave_sg()
  - Put the transaction in the driver pending queue using dmaengine_submit()
  - Ask the driver to process all pending transactions using dma_async_issue_pending()

https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Characteristics of I/O devices

- I/O devices
  - Block I/O
  - Character I/O (stream)
  - Memory-mapped file access
  - Network sockets
- Direct manipulation of I/O device
  - Linux's **ioctl** call that sends commands to a device driver

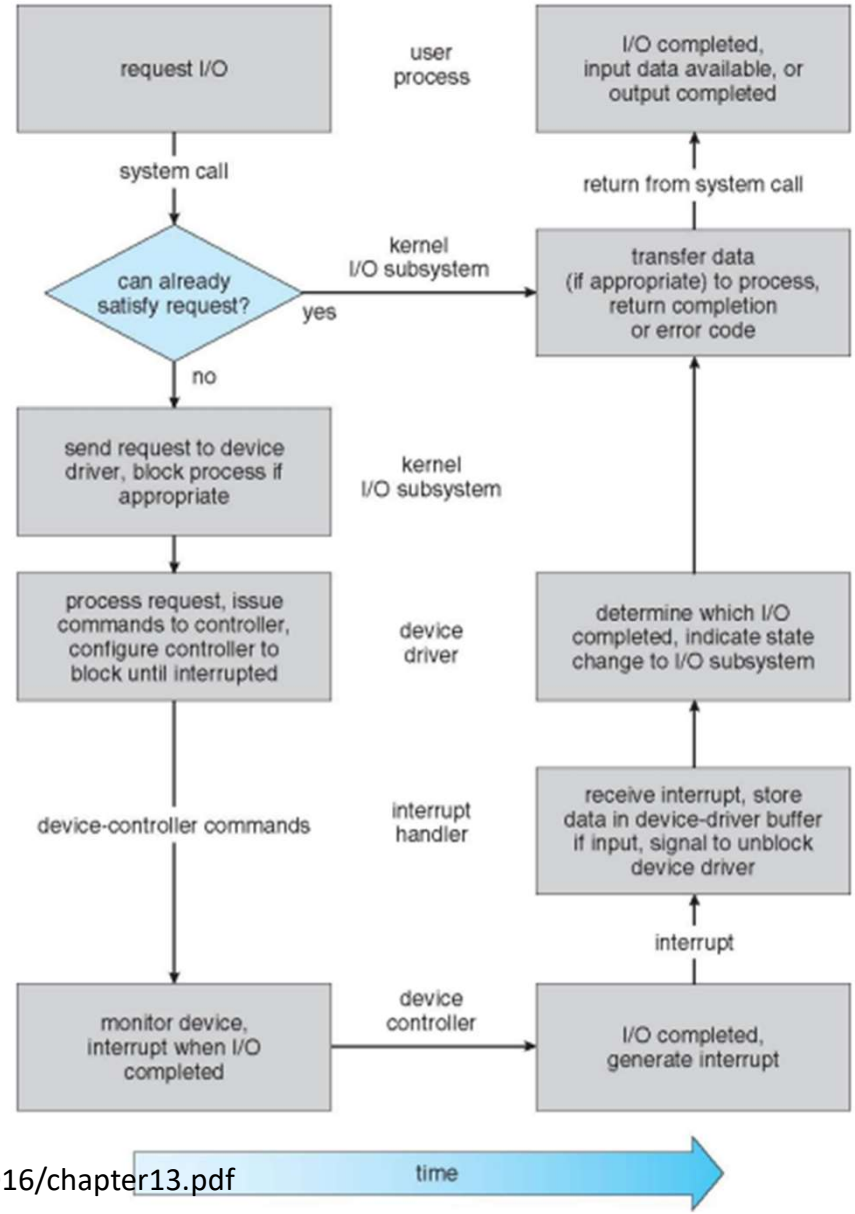
Block and character devices

- **Block devices** access data in blocks such as disk drives
  - Commands include read, write, seek
  - Raw I/O, direct I/O, or file system access
  - Memory-mapped file access possible
  - DMA
- **Character devices** include keyboards, mice, serial ports …

Synchronous/asynchronous I/O

- **Synchronous I/O**
  - **Blocking I/O:** process suspended until I/O completed
    - Simple, but less efficient
  - **Non-blocking I/O:** I/O calls return as much data as available
    - Process returns whatever existing data
    - Use to find if data is ready, then read or write to transfer data
- **Asynchronous I/O**
  - Process runs while I/O executes
  - I/O subsystem signals process when I/O completed via signal
  - Difficult to use but efficient

Life cycle of an I/O request



https://www.cs.fsu.edu/~zwang/files/cop4610/Fall2016/chapter13.pdf

Summary

- Approaches to communicate with I/O devices
  - Direct I/O instructions
  - Memory-mapped I/O
- Polling in programming I/O
- Interrupt-driven I/O
- Direct memory access(DMA)
  - Copy the data between devices and RAM without going through the CPU