Design and Analysis of Operating Systems CSCI 3753

Dr. David Knox
University of Colorado Boulder

Device Manager I/O Strategies

Design and Analysis of
Operating Systems
CSCI 3753

Dr. David Knox
University of Colorado
Boulder

Polling I/O: A Write Example

```
                                                    BUSY  DONE
while(deviceN.busy || deviceN.done) <waiting>;        *     *
                                                      0     0
deviceN.data[0] = <value to write>;

deviceN.command = WRITE;
                                                      1     0
while(deviceN.busy) <waiting>;
                                                      0     1
/* finished, so read some status bits
       for write operation … */

deviceN.done = FALSE;
                                                      0     0
```

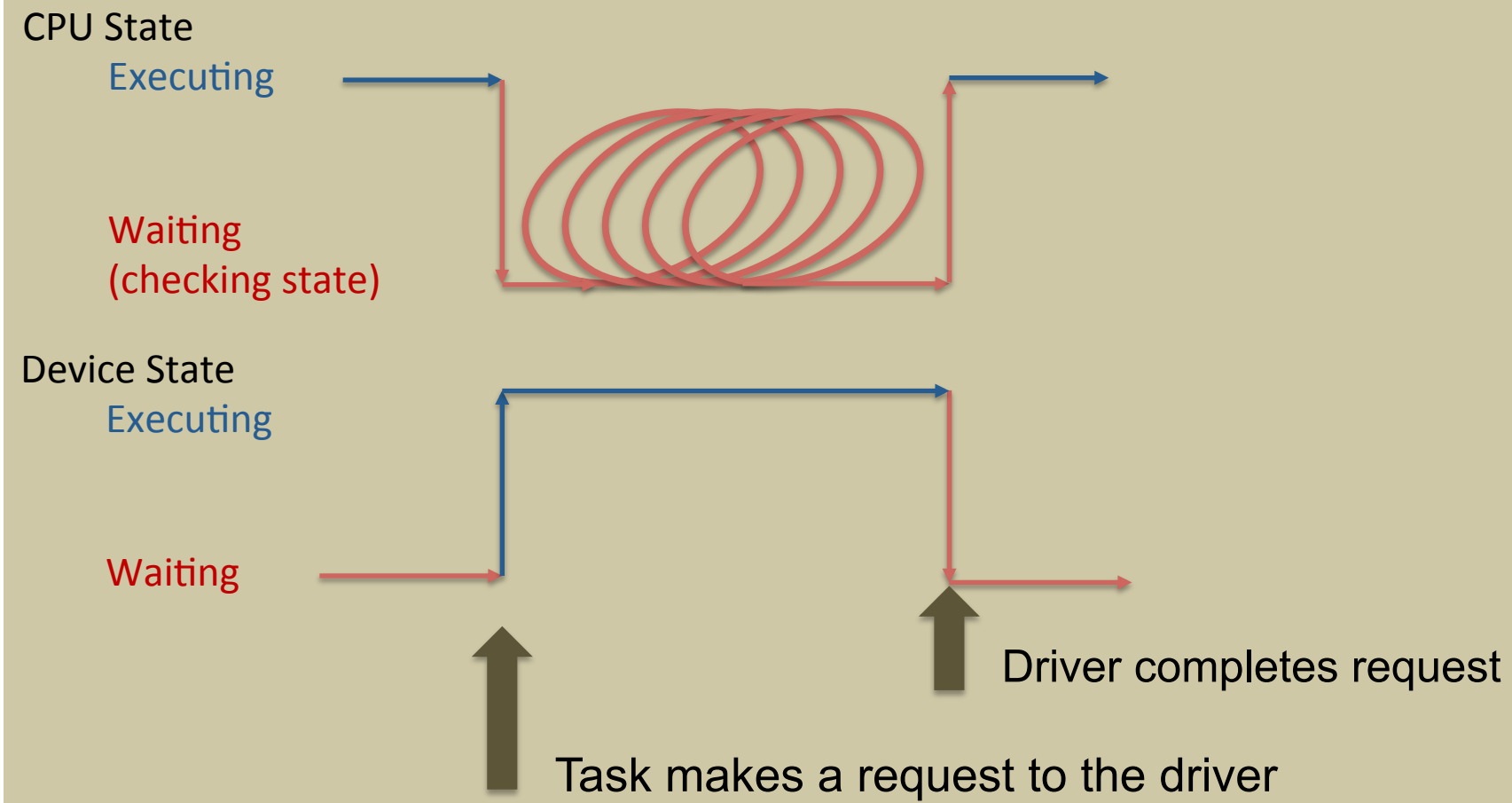
Polling I/O – Problem

- Busy waiting: this wastes CPU cycles that could be devoted to executing applications
  - To get the status of the device, the OS must spin in a loop waiting for the correct device state
- Instead, we can keep the CPU busy by *overlapping* CPU and I/O
  - Let the CPU do other work while the I/O device is processing a read/write

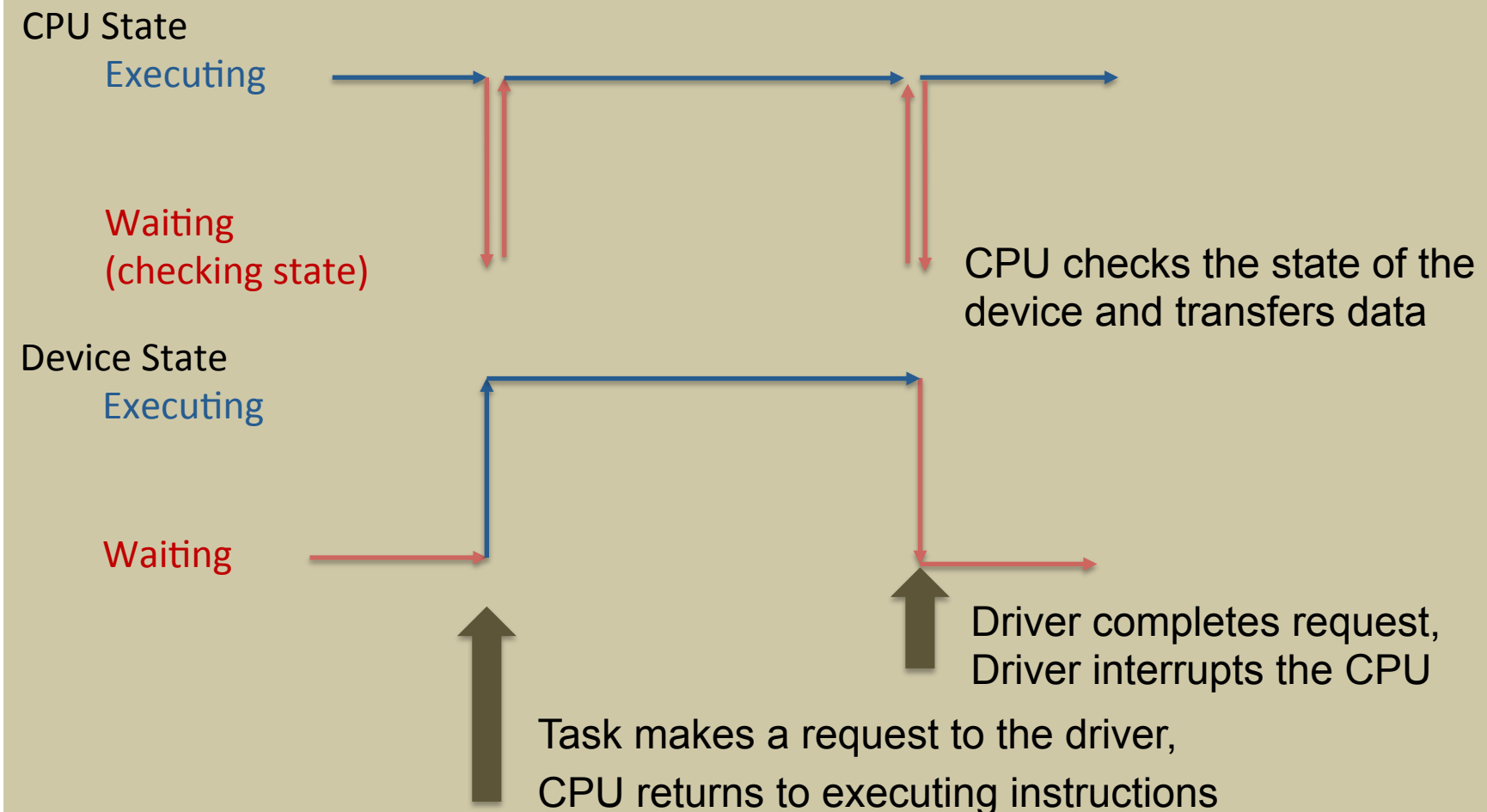
Device Manager I/O Strategies

- Underneath the system call API for a device, an OS can implement several strategies for I/O
  - direct I/O with *polling*
    - the OS device manager busy-waits (*previous example*)
  - direct I/O with *interrupts*
    - More efficient than busy waiting
  - DMA with interrupts

Direct I/O with *Polling*



Direct I/O with *Interrupts*



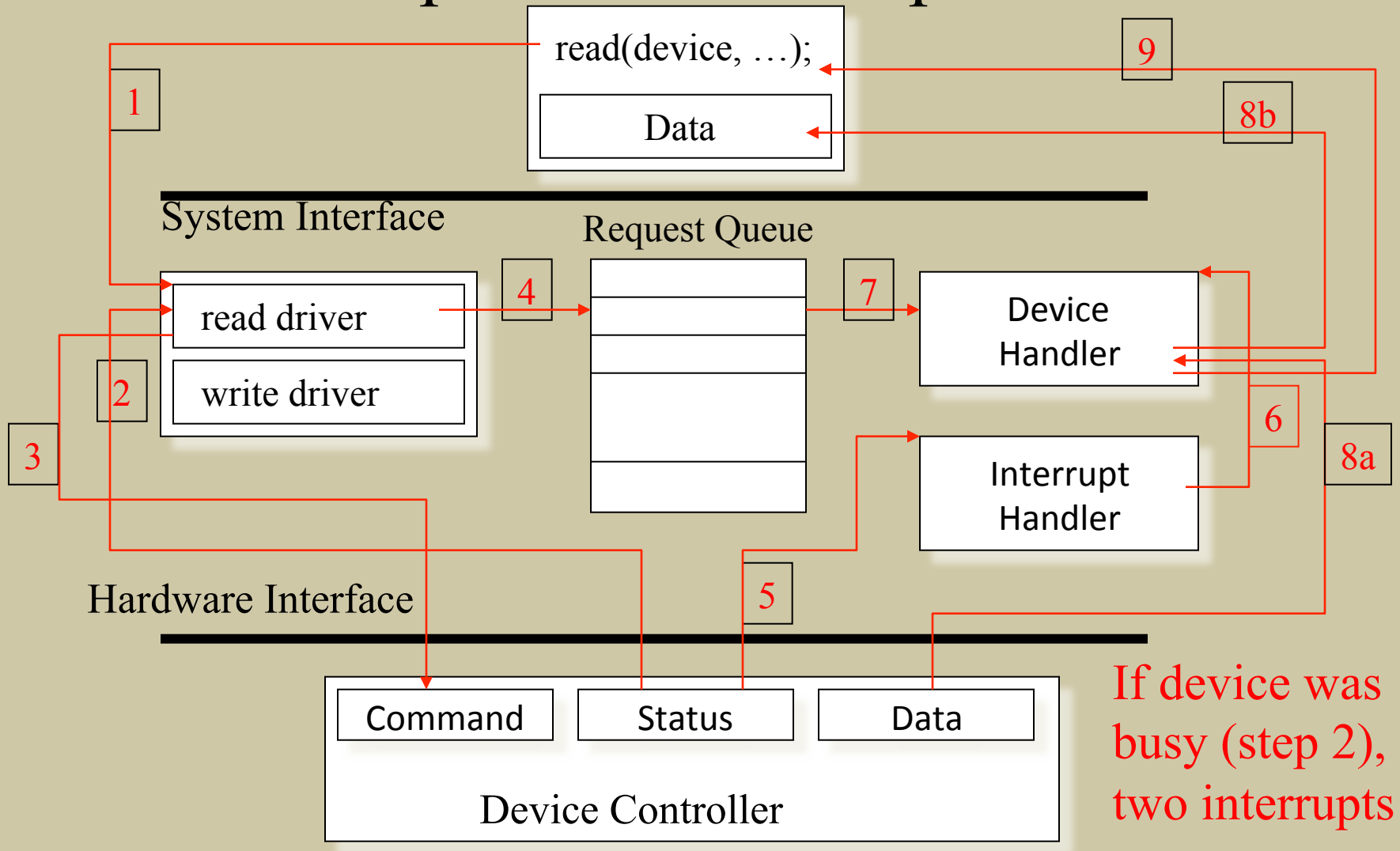
Direct I/O with *Interrupts*

- CPU incorporates a *hardware interrupt flag*
- Whenever a device is finished with a read/write, it communicates to the CPU and raises the flag
  - Frees up CPU to execute other tasks without having to keep polling devices
- Upon an interrupt, the CPU interrupts normal execution, and invokes the OS's *interrupt handler*
  - Eventually, after the interrupt is handled and the I/O results processed, the OS resumes normal execution

Interrupt Handler

- First, save the processor state
  - Save the executing task's program counter (PC) and CPU register data
- Next, find the device causing the interrupt
  - Consult interrupt controller to find the interrupt offset, or poll the devices
- Then, jump to the appropriate device handler
  - Index into the Interrupt Vector using the interrupt offset
  - An Interrupt Service Routine (ISR) either refers to the interrupt handler, or the device handler
- Finally, reenable interrupts

Interrupt-Driven I/O Operation



If device was busy (step 2), two interrupts occur

Why might *Polling* be BETTER than *Interrupt* handling?

- Setting up the interrupts takes overhead
- Handling the interrupts takes overhead
- Handling the scheduling of the processes takes overhead

Problem with Interrupt driven I/O

- Data transfer from disk can become a bottleneck if there is a lot of I/O copying data back and forth between memory and devices
  - Example: read a 1 MB file from disk into memory

    The disk is only capable of delivering 1 KB blocks

    So every time a 1 KB block is ready to be copied, an interrupt is raised, interrupting the CPU

    This slows down execution of normal programs and the OS

  - Worst case: CPU could be interrupted after the transfer of every byte/character, or every packet from the network card

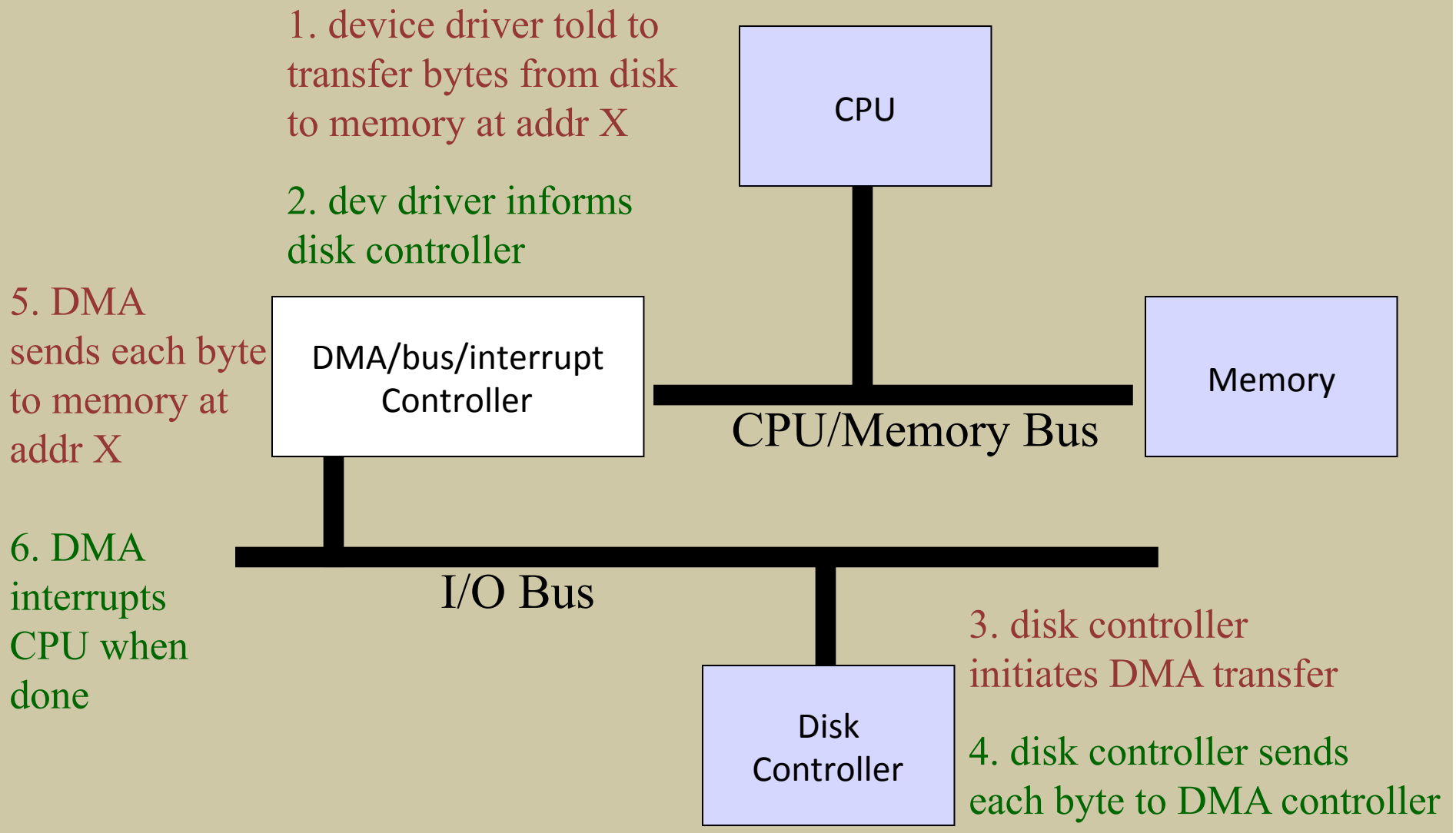
Device Manager I/O Strategies

- Underneath the system call API for a device, an OS can implement several strategies for I/O
  - direct I/O with polling
    - the OS device manager busy-waits (*previous example*)
  - direct I/O with *interrupts*
    - More efficient than busy waiting
  - ***DMA with interrupts***

Direct Memory Access (DMA)

- Bypass the CPU for large data copies, and only raise an interrupt at the very end of the data transfer, instead of at every intermediate block
- Modern systems offload some of this work to a special-purpose processor, Direct-Memory-Access (DMA) controller
- The DMA controller operates the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU

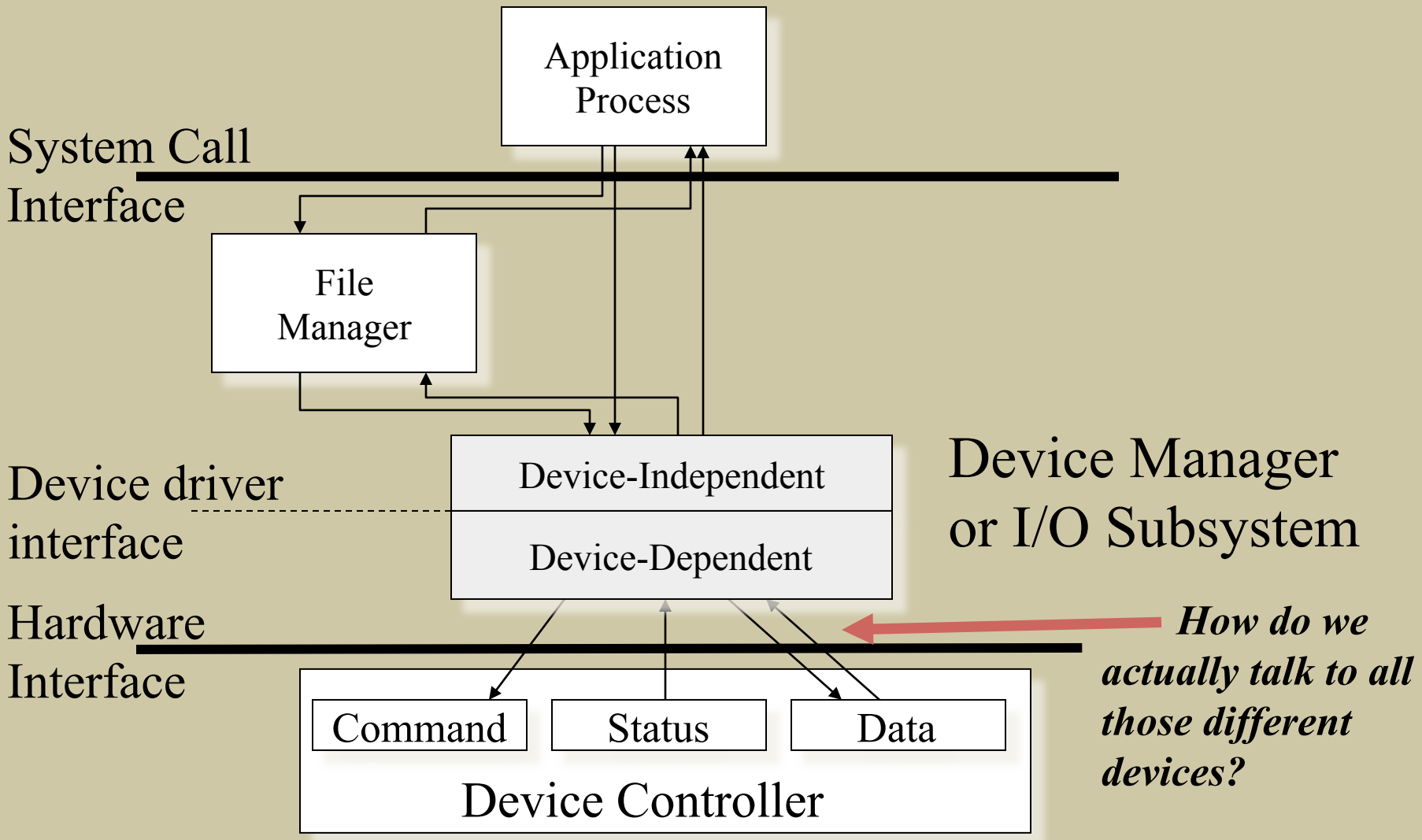
DMA with Interrupts Example



Direct Memory Access (DMA)

- Since both CPU and the DMA controller have to move data to/from main memory, how do they share main memory?
  - Burst mode
    - While DMA is transferring, CPU is blocked from accessing memory
  - Interleaved mode or "cycle stealing"
    - DMA transfers one word to/from memory, then CPU accesses memory, then DMA, then CPU, etc… - interleaved
  - Transparent mode – DMA only transfers when CPU is not using the system bus
    - Most efficient but difficult to detect

Device Management Organization



Port-Mapped I/O

- Port or port-mapped (non-memory mapped) I/O typically requires special I/O machine instructions to read/write from/to device controller registers
  - e.g. on Intel x86 CPUs, have IN, OUT
    - Example: OUT dest, src (using Intel syntax, not Gnu syntax)
      - Writes to a device port dest from CPU register src
    - Example: IN dest, src
      - Reads from a device port src to CPU register src
    - Only OS in kernel mode can execute these instructions
    - Later Intel introduced INS, OUTS (for strings), and INSB/INSW/INSD (different word widths), etc.

Device I/O Port Locations on PCs (partial)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

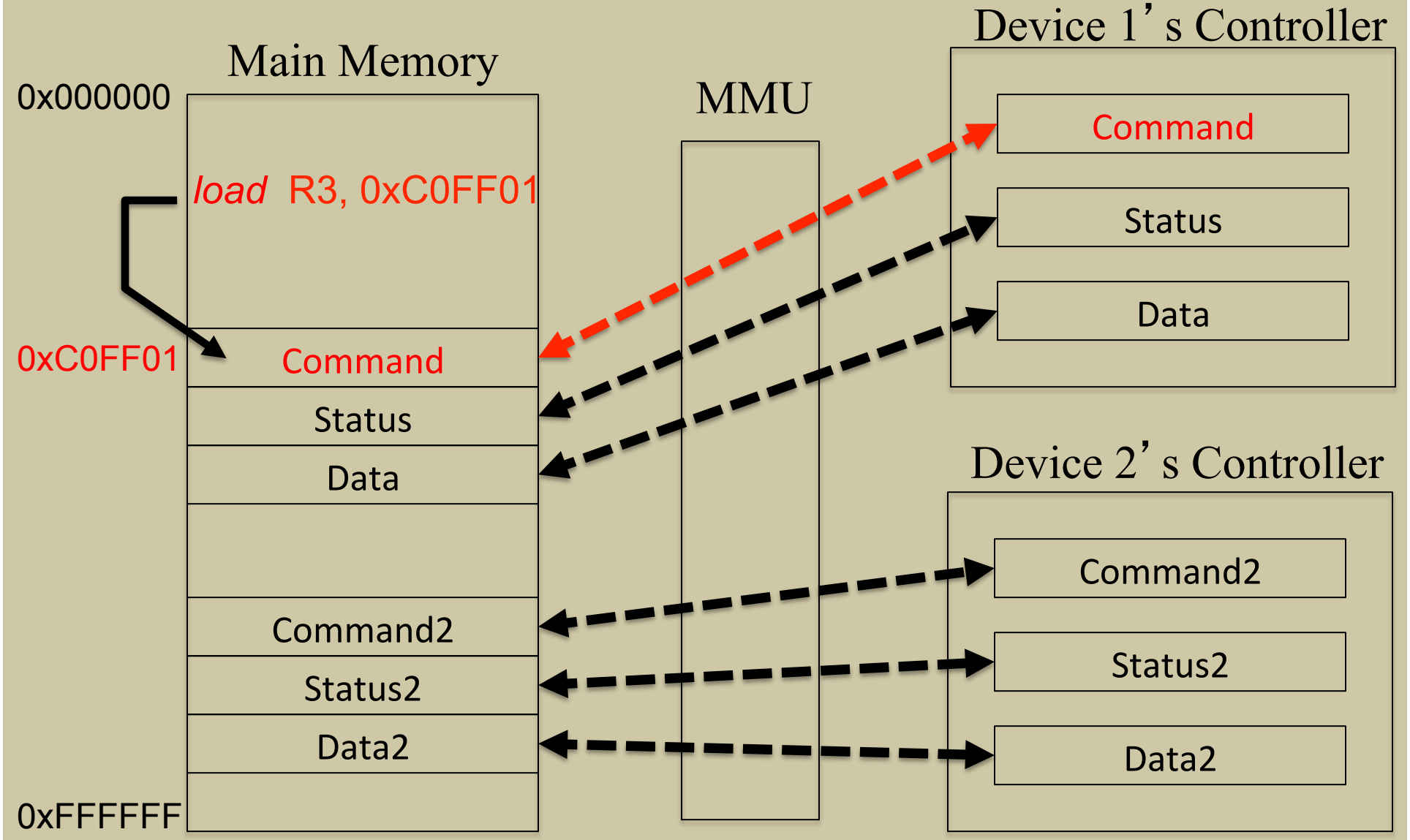
Port-Mapped I/O

- Port-mapped I/O is quite limited
  - IN and OUT can only store and load
  - don't have full range of memory operations for normal CPU instructions
    - Example: to increment the value in say a device's data register, have to copy register value into memory, add one, and copy it back to device register.
    - AMD did not extend the port I/O instructions when defining the x86-64

Memory-Mapped I/O

- Memory-mapped I/O: device registers and device memory are mapped to the system address space
- With memory-mapped I/O, just address memory directly using normal instructions to speak to an I/O address
  - e.g. load R3, 0xC0FF01
  - the memory address 0xC0FF01 is mapped to an I/O device's register
- Memory Management Unit (MMU) maps memory values and data to/from device registers
  - Device registers are assigned to a block of memory
  - When a value is written into that I/O-mapped memory, the device sees the value, loads the appropriate value and executes the appropriate command

Memory-Mapped I/O



Memory-Mapped I/O

- Typically, devices are mapped into lower memory
  - frame buffers for displays take the most memory, since most other devices have smaller buffers
  - Even a large display might take only 10 MB of memory, which in modern address spaces of tens-hundreds of GBs is quite modest – so memory-mapped I/O is a small penalty

What is difference between Port and Memory Mapped IO?

- **Port mapped I/O** uses a separate, dedicated address space and is accessed via a dedicated set of microprocessor instructions.

- **Memory mapped I/O** is **mapped** into the same address space as program **memory** and/or user **memory**, and is accessed in the normal way.

Design and Analysis of Operating Systems CSCI 3753

Dr. David Knox

University of Colorado
Boulder