Device Management

COMP 3361: Operating Systems I

Winter 2015

http://www.cs.du.edu/3361

1 I/O Devices

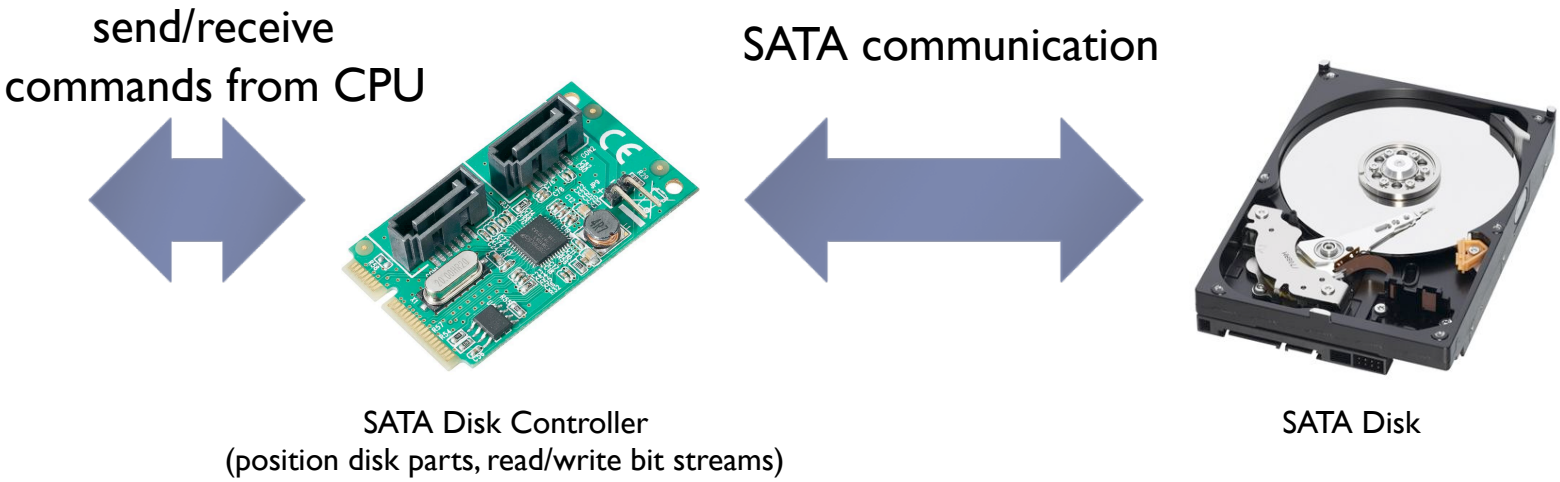
- Block devices
  - stores information in fixed-size blocks
  - transfers are in units of entire blocks
  - example: hard disk
- Character devices
  - delivers or accepts stream of characters, without regard to block structure
  - not addressable, does not have any *seek* operation
  - example: printer
- Others
  - example: clocks

2 I/O Device Speed

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner at 300 dpi	1 MB/sec
Digital camcorder	3.5 MB/sec
4x Blu-ray disc	18 MB/sec
802.11n Wireless	37.5 MB/sec
USB 2.0	60 MB/sec
FireWire 800	100 MB/sec
Gigabit Ethernet	125 MB/sec
SATA 3 disk drive	600 MB/sec
USB 3.0	625 MB/sec
SCSI Ultra 5 bus	640 MB/sec
Single-lane PCIe 3.0 bus	985 MB/sec
Thunderbolt 2 bus	2.5 GB/sec
SONET OC-768 network	5 GB/sec

Device Controller

- An electronic component that talks to the device
- Language is standardized
  - SATA, SCSI, USB, Thunderbolt, …
  - device can be built independent of controller

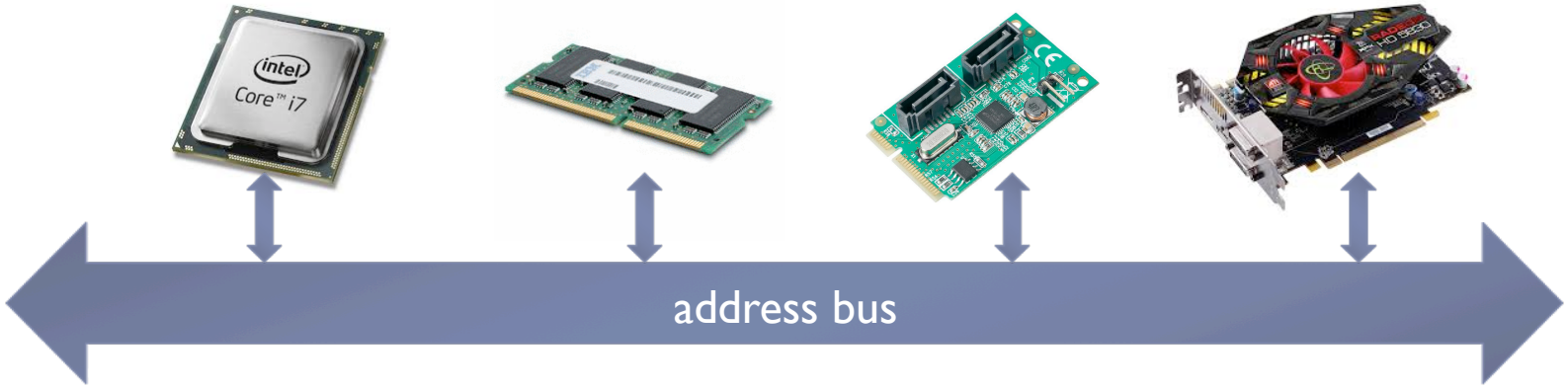


4 Talking to a Controller

- Each controller has a few control registers
  - controller responds to data (commands) written to the registers
- Controller can also has data registers and buffers
  - used to hold data intermediately during transit from memory to device
  - OS can read from the data buffers
  - devices can also have buffers; OS can trigger controller to read device buffer to memory

Talking to a Controller

- **I/O port**: controller registers are assigned specific numbers
  - OS can read in and write to the controller's registers using the numbers
- **Memory-mapped I/O**: controller registers/buffers are mapped to specific physical memory address range
  - OS reads in and writes to memory region
  - controller writes to and reads in from memory region



Memory-Mapped I/O

- Can communicate with device controllers with memory read/write
  - no assembly coding necessary
- OS can allow a user program to control a device
  - set up page tables accordingly
- All instructions that can access memory can access devices
  - can easily check state of registers; instead of first reading it to a memory location and then checking
- Must be able to selectively disable memory caching

7 Reading From Disk

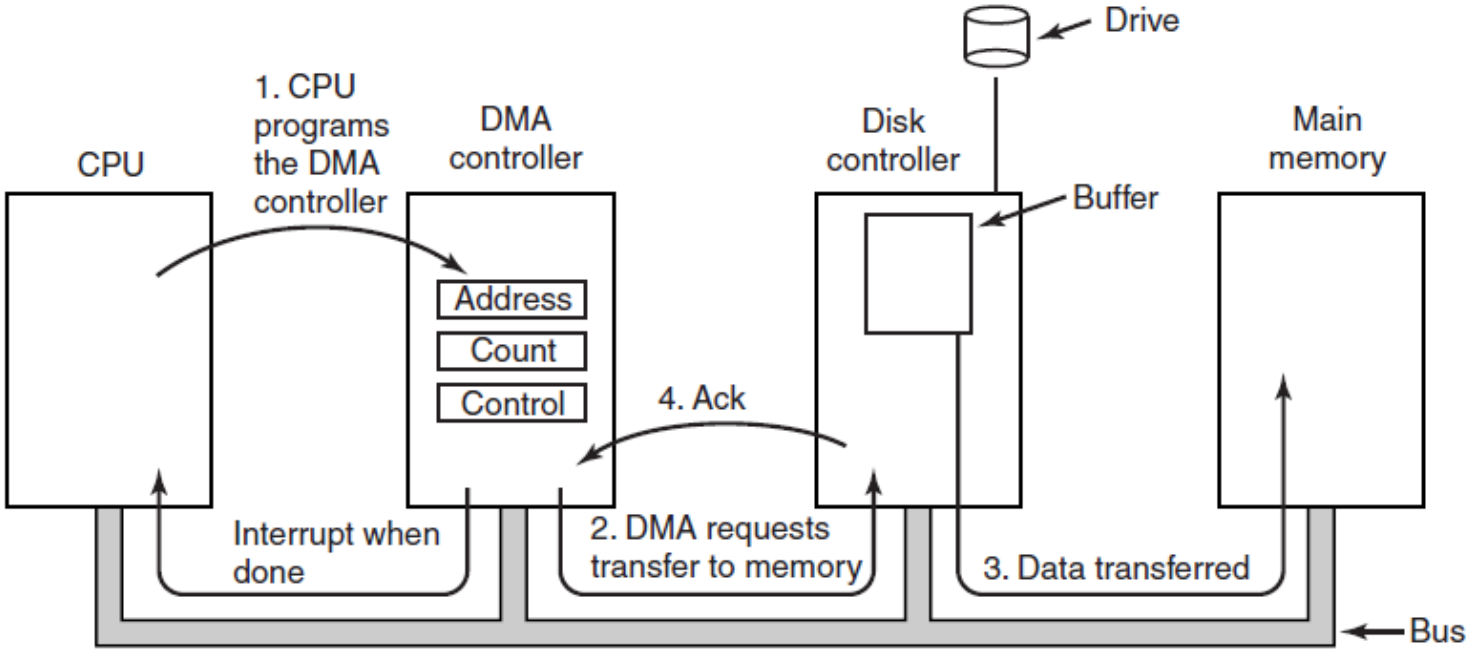
```
uint8_t read_disk(uint32_t LBA, uint8_t n_sectors, uint8_t *buffer) {
      uint8_t status;
      int i;
      uint16_t sectors_to_read;
      uint16_t *data = (uint16_t *)buffer;
      ...
      // LBA mode (bit 6) and highest four bits of LBA (bit 7 and 5 are always set)
      port_write_byte(0x1F6, 0xE0 | ((LBA >> 24) & 0x0F));

      port_write_byte(0x1F1,0x00);           // NULL byte
      port_write_byte(0x1F2,n_sectors);           // sector count
      port_write_byte(0x1F3,(uint8_t)LBA);        // low 8 bits of LBA
      port_write_byte(0x1F4,(uint8_t)(LBA>>8));   // next 8 bits of LBA
      port_write_byte(0x1F5,(uint8_t)(LBA>>16));  // next 8 bits of LBA
      port_write_byte(0x1F7,0x20);           // send READ SECTORS command

      sectors_to_read = (n_sectors==0)?256:n_sectors;

      for (; sectors_to_read>0; sectors_to_read--) {
          // poll for readiness
          ...
          // read one sector
          for(i=0; i<256; i++) {
               data[i] = port_read_word(0x1F0); // read one word (2 bytes)
          }
          data += 256;
          ...
      }
      return NO_ERROR;
}
```

Using a DMA Controller



Interrupt Controller

- Interrupts generated by devices are read by the interrupt controller
  - devices assert a signal on an assigned interrupt line
  - controller looks out for these signals
- Interrupt controller puts a number on address lines and asserts the interrupt line going to the CPU
- Service routine notifies controller of service completion by writing to a special port of the controller
  - controller can notify device of service completion and attend to a pending interrupt from another device

Programmed I/O

- CPU determines if device is available and issues commands to controller for every byte/word it needs to read/write
  - example: slide 7

```
copy_from_user(buffer, p, count);                /* p is the kernel buffer */
for (i = 0; i < count; i++) {                    /* loop on every character */
        while (*printer_status_reg != READY) ;   /* loop until ready */
        *printer_data_register = p[i];           /* output one character */
}
return_to_user( );
```

printing using programmed I/O

Interrupt-Driven I/O

- Remove polling or busy waiting
  - is the device ready for next command? …
- Can be used if controllers can (be programmed to) generate an interrupt on command completion
  - controller notifies CPU when command completes
  - CPU then carries out next command

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

printing first character

```
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count – 1;
     i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

printer interrupt service routine

I/O Using DMA

- Interrupt-drive I/O requires interrupt handling for every byte/word of data
- Programmed I/O, but with hardware (DMA) support
  - let DMA controller talk to device controller and transfer data (whatever the DMA controller's buffer can hold)
  - DMA controller issues interrupt to CPU when buffer is full

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

set up DMA

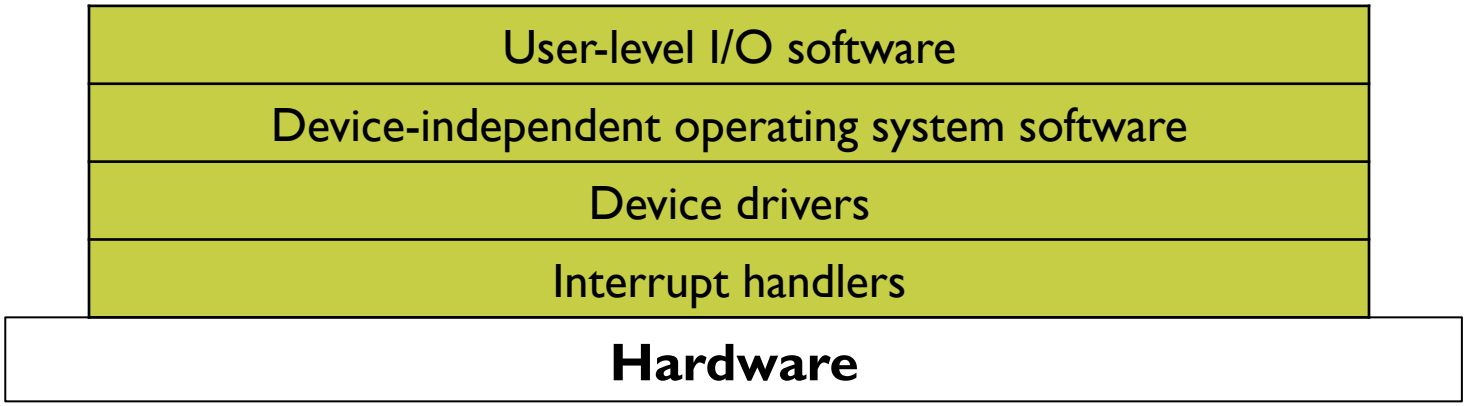
```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

DMA interrupt service routine

(assuming DMA buffer is big enough to hold all data)

I/O Software Layers

- What software layers are involved in accessing hardware?



14 Interrupt Handler

- Interrupts are mechanisms for controllers to inform CPU of an event ("I finished the task you assigned")
- Interrupt handlers run when interrupts are generated

- OS responsibilities
  - save process state
  - set up execution environment for handler
  - run handler
  - restore state

Device Driver

- Device controllers expect commands from the CPU
- What commands will do what?
  - specific to device
- **Device driver**: program that knows what commands to issue to extract a specific functionality from a specific device controller
  - e.g.: command to obtain disk size from a SATA disk controller
- Runs as part of the kernel
  - OS can implement drivers
  - allow for their installation; OS defines standard interface that it will use to interact with the drivers

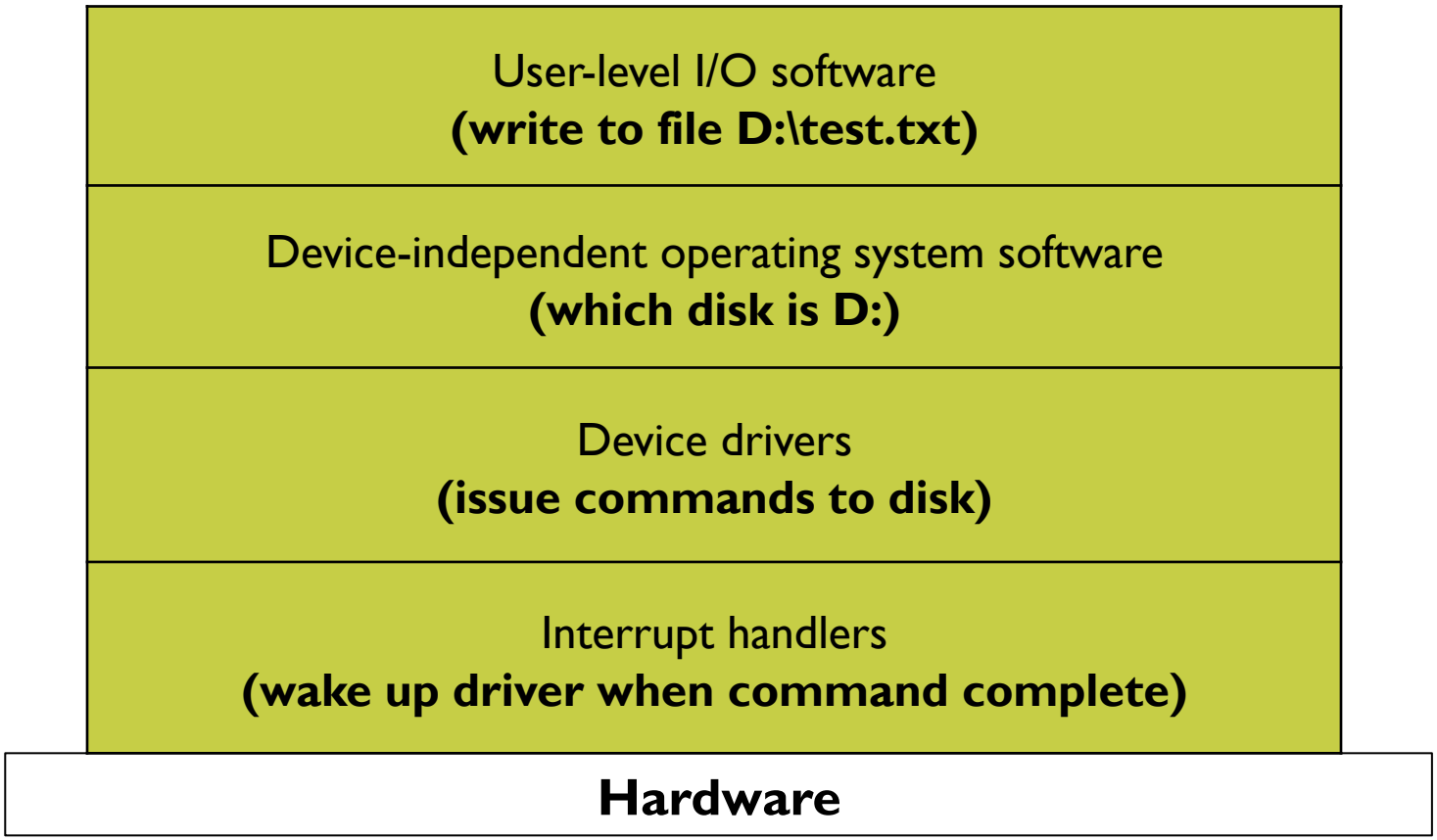
16 Device-Independent I/O Software

- Perform I/O functions common to all devices
- Provide common interface to user programs
- Uniform interface
  - device driver
  - I/O device naming (e.g. D: or /dev/disk0)
  - device protection
- Buffering: making data available in expected sizes
- Error reporting
- Managing requests for devices

User-Space I/O Software

- System calls that allow interaction with device
  - e.g. read, write, print, scanf
- Implemented as part of a library
- May also perform formatting or spooling
  - formatting: `printf("[%d,%d]",i,j)`
  - spooling: collect data from multiple processes and handle one by one

I/O Software Layers Example



References

- Chapter 5.1-5.3, Modern Operating Systems, A. Tanenbaum and H. Bos, 4th Edition.

What Next?

- Finish reading the partially covered chapters

- Chapter 8: How basic concepts differ in multiple processor systems?
- Case studies: How real world operating systems implement the basic concepts?
  - Chapter 10: Linux
  - Chapter 11: Windows