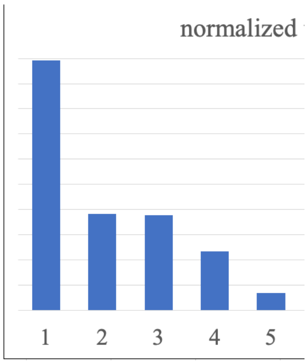
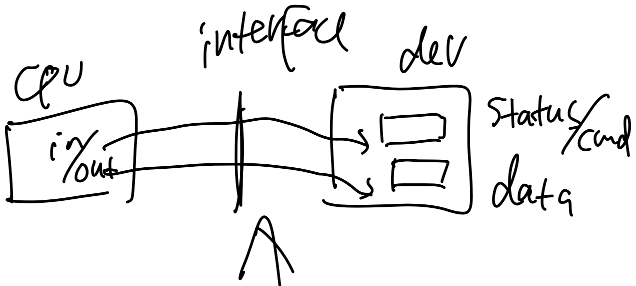
Week 12.a
CS 5600
03/27 2023

1. I/O (continued)
2. device driver
3. Disks
4. SSDs



•



CPU-I/O interactions, four approaches:
 * port-mapped I/O
 * memory-mapped I/O
 * interrupts
 * via memory

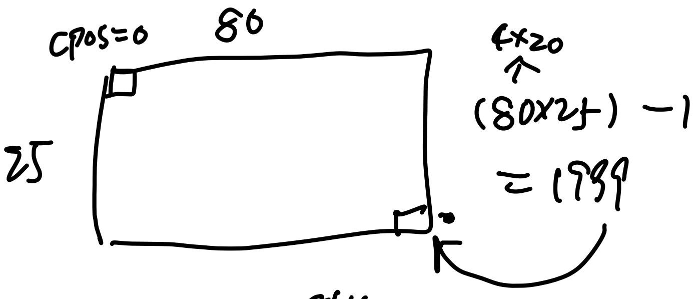
```
CS5600 Week11.b

1. Two examples of I/O instructions

  (a) Reading keyboard input

    The code below is an excerpt from WeensyOS.
    (details in PS/2 controller: https://wiki.osdev.org/%228042%22_PS/2_Controller)
    This reads a character typed at the keyboard (which shows up on the
    "keyboard data port" (kEYBOARD_DATAREG)).

    /* Excerpt from WeensyOS x86-64.h and k-hardware.cc */
    // Keyboard programmed I/O
    #define KEYBOARD_STATUSREG 0x64
    #define KEYBOARD_STATUS_READY 0x01
    #define KEYBOARD_DATAREG 0x60

    int keyboard_readc() {
      static uint8_t modifiers;
      static uint8_t last_escape;

      if ((inb(KEYBOARD_STATUSREG) & KEYBOARD_STATUS_READY) == 0) {
        return -1;
      }

      uint8_t data = inb(KEYBOARD_DATAREG);
      uint8_t escape = last_escape;
      last_escape = 0;

      if (data == 0xE0) { // mode shift
        last_escape = 0x80;
        return 0;
      } else if (data & 0x80) { // key release: matters only
                                //  for modifier keys
        int ch = keymap[(data & 0x7F) | escape];
        if (ch >= KEY_SHIFT && ch < KEY_CAPSLOCK) {
          modifiers &= ~(1 << (ch - KEY_SHIFT));
        }
        return 0;
      }

      int ch = (unsigned char) keymap[data | escape];

      if (ch >= 'a' && ch <= 'z') {
        if (modifiers & MOD_CONTROL) {
          ch -= 0x60;
        } else if (!(modifiers & MOD_SHIFT) != \
                   !(modifiers & MOD_CAPSLOCK)) {
          ch -= 0x20;
        }
      } else if (ch >= KEY_CAPSLOCK) {
        modifiers ^= 1 << (ch - KEY_SHIFT);
        ch = 0;
      } else if (ch >= KEY_SHIFT) {
        modifiers |= 1 << (ch - KEY_SHIFT);
        ch = 0;
      } else if (ch >= CKEY(0) && ch <= CKEY(21)) {
        ch = complex_keymap[ch - CKEY(0)].map[modifiers & 3];
      } else if (ch < 0x80 && (modifiers & MOD_CONTROL)) {
        ch = 0;
      }

      return ch;
    }
```

port

```
(b) Setting the cursor position

  The code below is also excerpted from WeensyOS. It uses I/O
  instructions to set a blinking cursor. To set the cursor to
  the upper left of the screen, run:  console_show_cursor(0)

  // console_show_cursor(cpos)
  //   Move the console cursor to position ‘cpos‘,
  //   which should be between 0 and 80 * 25.

  void console_show_cursor(int cpos) {
    if (cpos < 0 || cpos > CONSOLE_ROWS * CONSOLE_COLUMNS)
      cpos = 0;

   outb(0x3D4, 14); // ~~Command 14 = upper byte of position~~
   outb(0x3D5, cpos / 256); // upper byte (256 = 2^8)
   outb(0x3D4, 15); // Command 15 = lower byte of position
   outb(0x3D5, cpos % 256); // lower byte

  }

  // if interested, see details: https://wiki.osdev.org/Text_Mode_Cursor
```

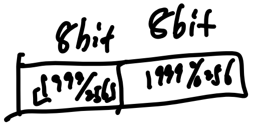
1999

20000



8bit

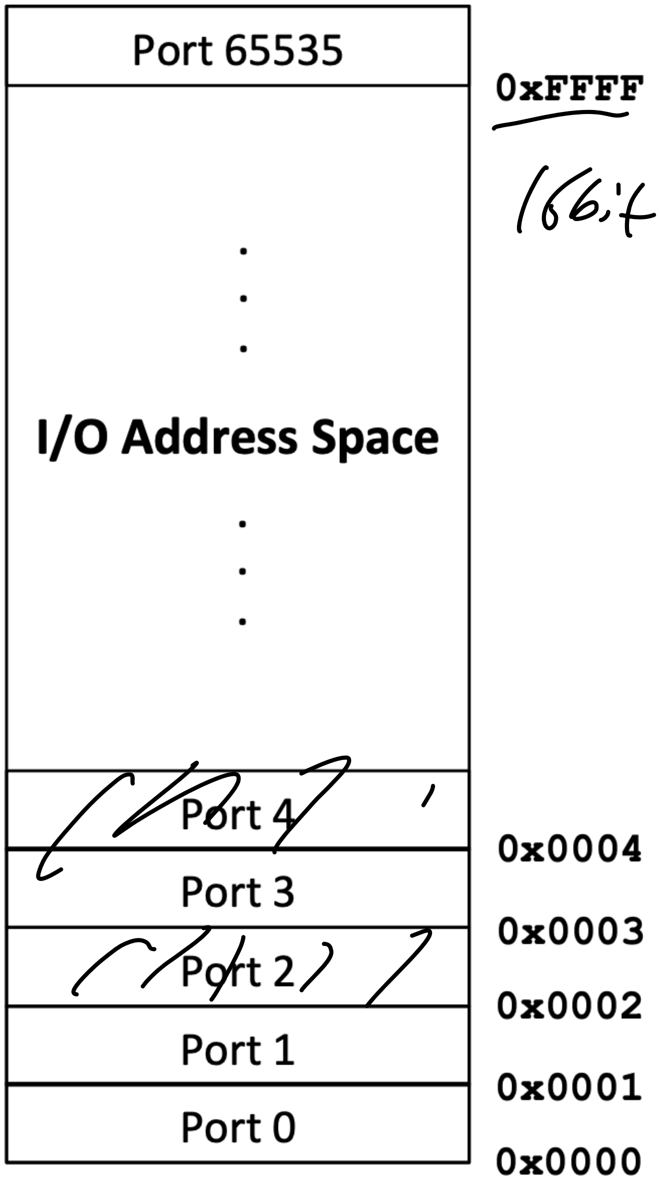
$2^8 = 256$



(PM20)

Port I/O Address Space

- Software and hardware architectures of x86 architecture support a separate address space called “I/O Address Space”
  - Separate from memory space
- Access to this separate I/O space is handled through a set of I/O instructions
  - IN,OUT, INS, OUTS
- Access requires Ring0 privileges
  - Access requirement does not apply to all operating modes (like Real-Mode)
- The processor allows 64 KB+3 bytes to be addressed within the I/O space
- Harkens back to a time when memory was not so plentiful
- You may never see port I/O when analyzing high-level applications, but in systems programming (and especially BIOS) you will see lots of port I/O
- One of the biggest impediments to understanding what's going on in a BIOS



Port 65535

0xFFFF

16bit

I/O Address Space

Port 4

0x0004

Port 3

0x0003

Port 2

0x0002

Port 1

0x0001

Port 0

0x0000

Intel Programmer’s guide, Vol 1, 16.1

```
2. Memory-mapped I/O

  (a) Here is a 32-bit PC’s physical memory map:

      +------------------+ <- 0xFFFFFFFF (4GB)
      |      32-bit      |
      |  memory mapped   |
      |     devices      |
      |                  |
      /\/\/\/\/\/\/\/\/\/\

      /\/\/\/\/\/\/\/\/\/\
      |                  |
      |      Unused      |
      |                  |
      +------------------+ <- depends on amount of RAM
      |                  |
      |                  |
      | Extended Memory  |
      |                  |
      |                  |
      +------------------+ <- 0x00100000 (1MB)
      |     BIOS ROM     |
      +------------------+ <- 0x000F0000 (960KB)
      |  16-bit devices, |
      |  expansion ROMs  |
      +------------------+ <- 0x000C0000 (768KB)
      |   VGA Display    |
      +------------------+ <- 0x000A0000 (640KB)
      |                  |
      |    Low Memory    |
      |                  |
      +------------------+ <- 0x00000000

  [Credit to Frans Kaashoek, Robert Morris, and
  Nickolai Zeldovich for this picture]
```

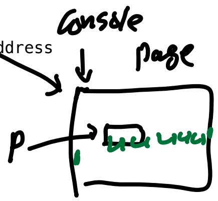
monitor

Registers

B800

```
(b) Loads and stores to the device memory "go to hardware".

  Here is an excerpt of the console printing code from WeensyOS.


  /* Compare the address below to the map in panel 2(a). */
  PROVIDE(console = 0xB8000);

  This is an excerpt about printing; notice how it uses the address
  "console":

  /*
  * prints a character to the console at the specified
  * cursor position in the specified color.
  * Question: what is going on in the check
  * if (c == '\n')
  * ?
  * Hint: '\n' is "newline" (the user pressed enter).
  */
  static void console_putc(printer* p, unsigned char c, int color) {
    console_printer* cp = (console_printer*) p;
    if (cp->cursor >= console + CONSOLE_ROWS * CONSOLE_COLUMNS) {
      cp->cursor = console;
    }
    if (c == '\n') {
      int pos = (cp->cursor - console) % 80;
      for (; pos != 80; pos++) {
        *cp->cursor++ = ' ' | color;
      }
    } else {
      *cp->cursor++ = c | color;
    }
  }
```

'a'



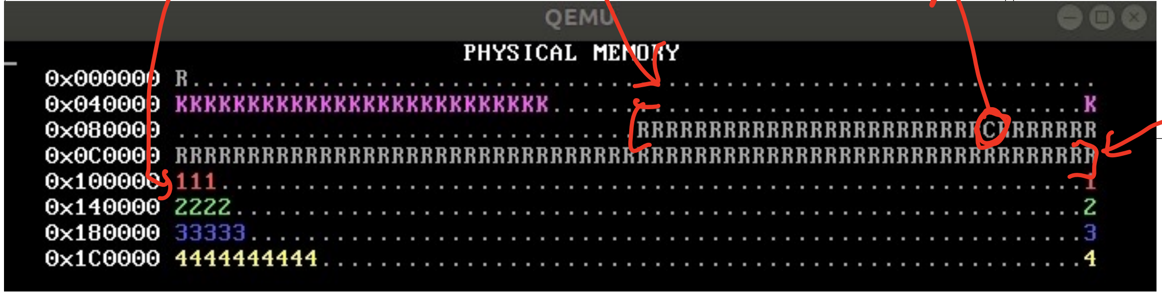
0x B8000

Console

Process

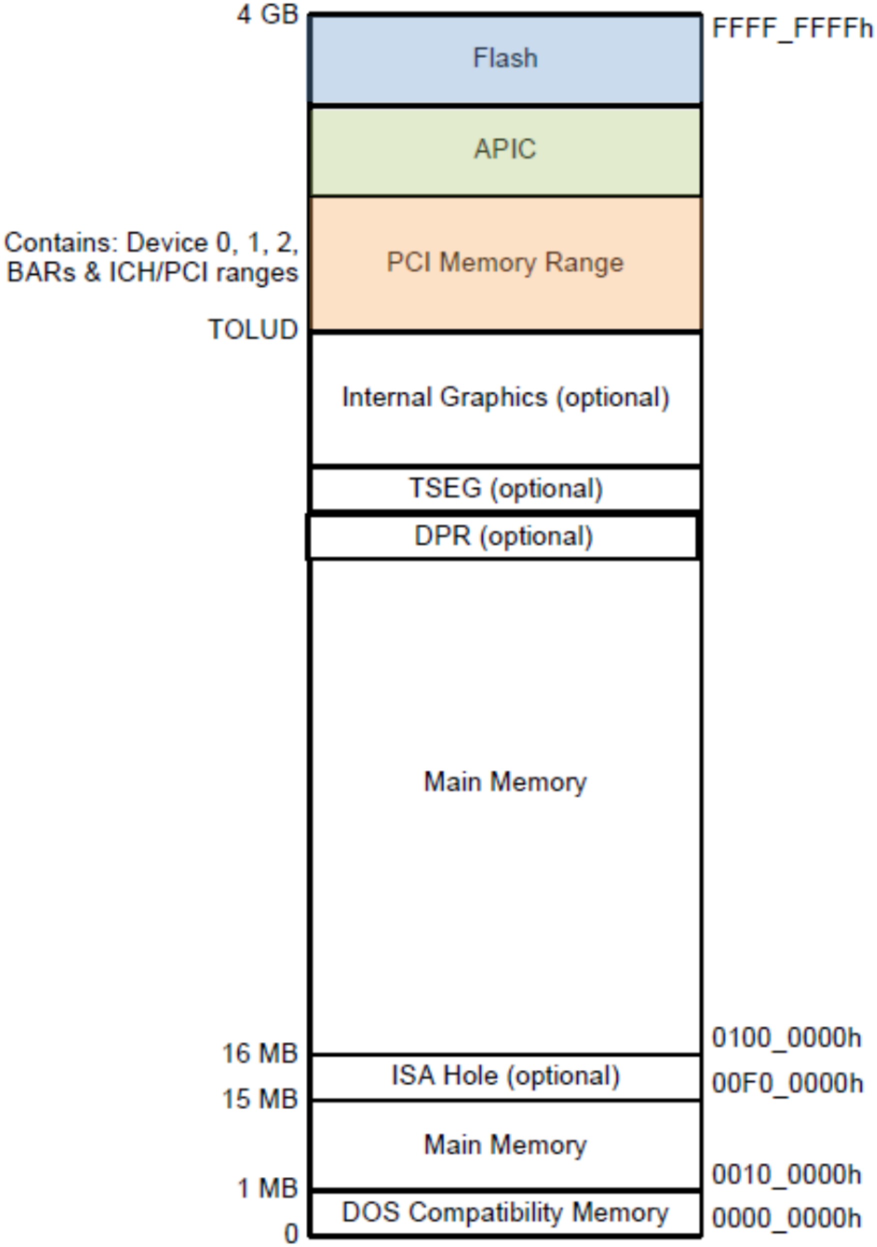
0x100000 - ε

?



```
                          PHYSICAL MEMORY
0x000000 R...............................................................
0x040000 KKKKKKKKKKKKKKKKKKKKKKKKKK.....................................K
0x080000 ................................RRRRRRRRRRRRRRRRRRRRRRRRCRRRRRRR
0x0C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
0x100000 111............................................................1
0x140000 2222...........................................................2
0x180000 33333..........................................................3
0x1C0000 4444444444.....................................................4
```

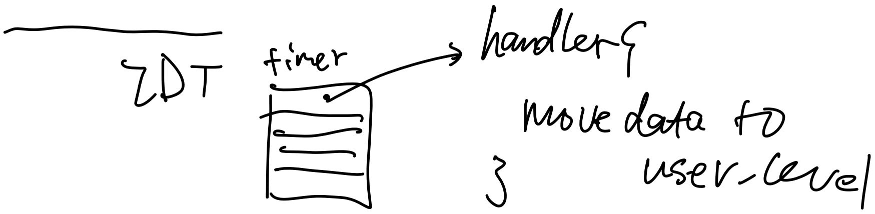
Memory Mapped IO (MMIO)



- The colored regions are memory mapped devices
- Accesses to these memory ranges are decoded to a device itself
- Flash refers to the BIOS flash
- APIC is the Advanced Programmable Interrupt Controller
- PCI Memory range is programmed by BIOS in the PCIEXBAR

- Polling vs. interrupts ← III

```
while ( 1 ) {
      inb(REG);
      check if Ready, if so break
}
```

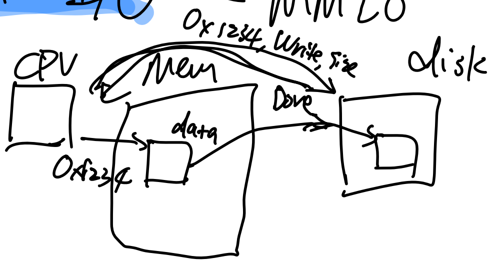


- DMA vs. programmed I/O

PMIO

MMIO

DMAC (hardware)



```
* a concrete but fake example
  (borrowed from https://www.xml.com/ldd/chapter/book/ch13.html)

   int dad_transfer(struct dad_dev *dev, int write, void *buffer,
                    size_t count)
   {
       dma_addr_t bus_addr;
       unsigned long flags;

       /* Map the buffer for DMA */
       dev->dma_dir = (write ? PCI_DMA_TODEVICE : PCI_DMA_FROMDEVICE);
       dev->dma_size = count;
       bus_addr = pci_map_single(dev->pci_dev, buffer, count,
                                 dev->dma_dir);
       dev->dma_addr = bus_addr;

       /* Set up the device */
       writeb(dev->registers.command, DAD_CMD_DISABLEDMA);
       writeb(dev->registers.command, write ? DAD_CMD_WR : DAD_CMD_RD);
       writel(dev->registers.addr, cpu_to_le32(bus_addr));
       writel(dev->registers.len, cpu_to_le32(count));
```

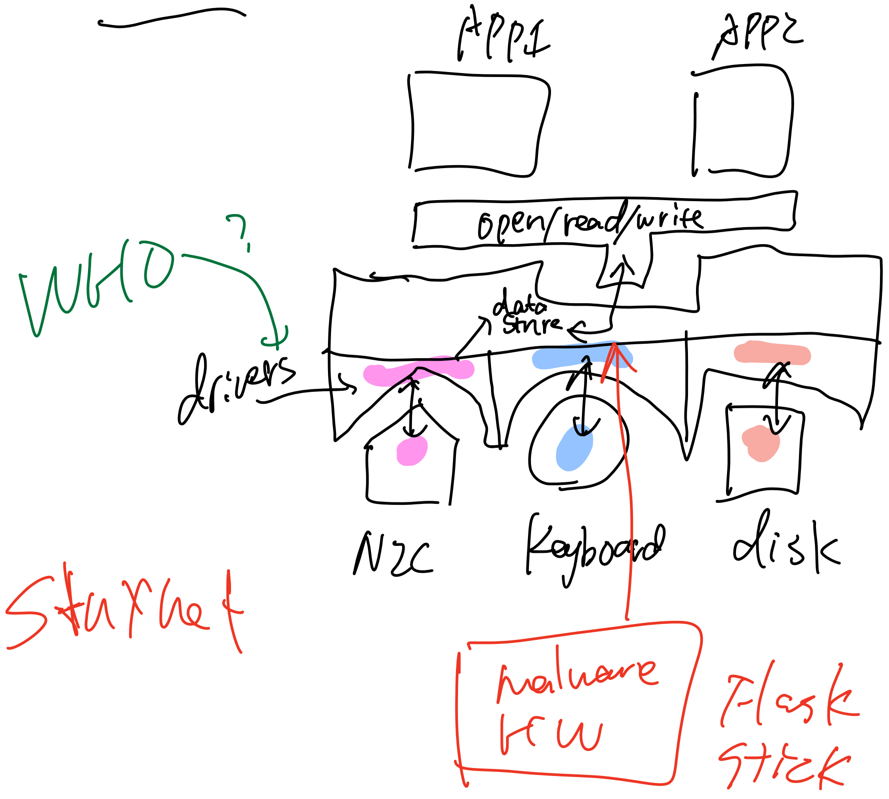
```
        /* Start the operation */
        writeb(dev->registers.command, DAD_CMD_ENABLEDMA);
        return 0;
    }
```

{polling, interrupts} ×

{DMA, PIO}

PMIO MMIO

* Device drivers



App1
App2
open/read/write
data store
WHO ?
drivers
NIC
Keyboard
disk
Stuxnet
malware F/W
Flash stick