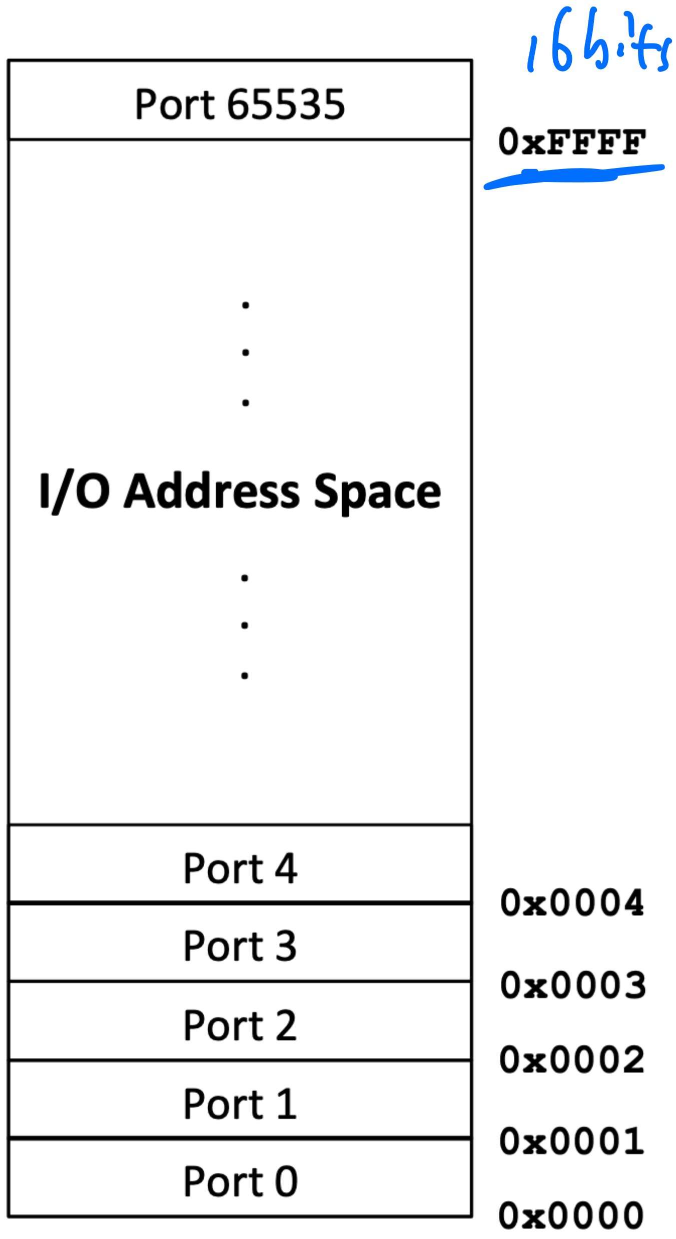
Port I/O Address Space

- 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



16bits

Port 65535

0xFFFF

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

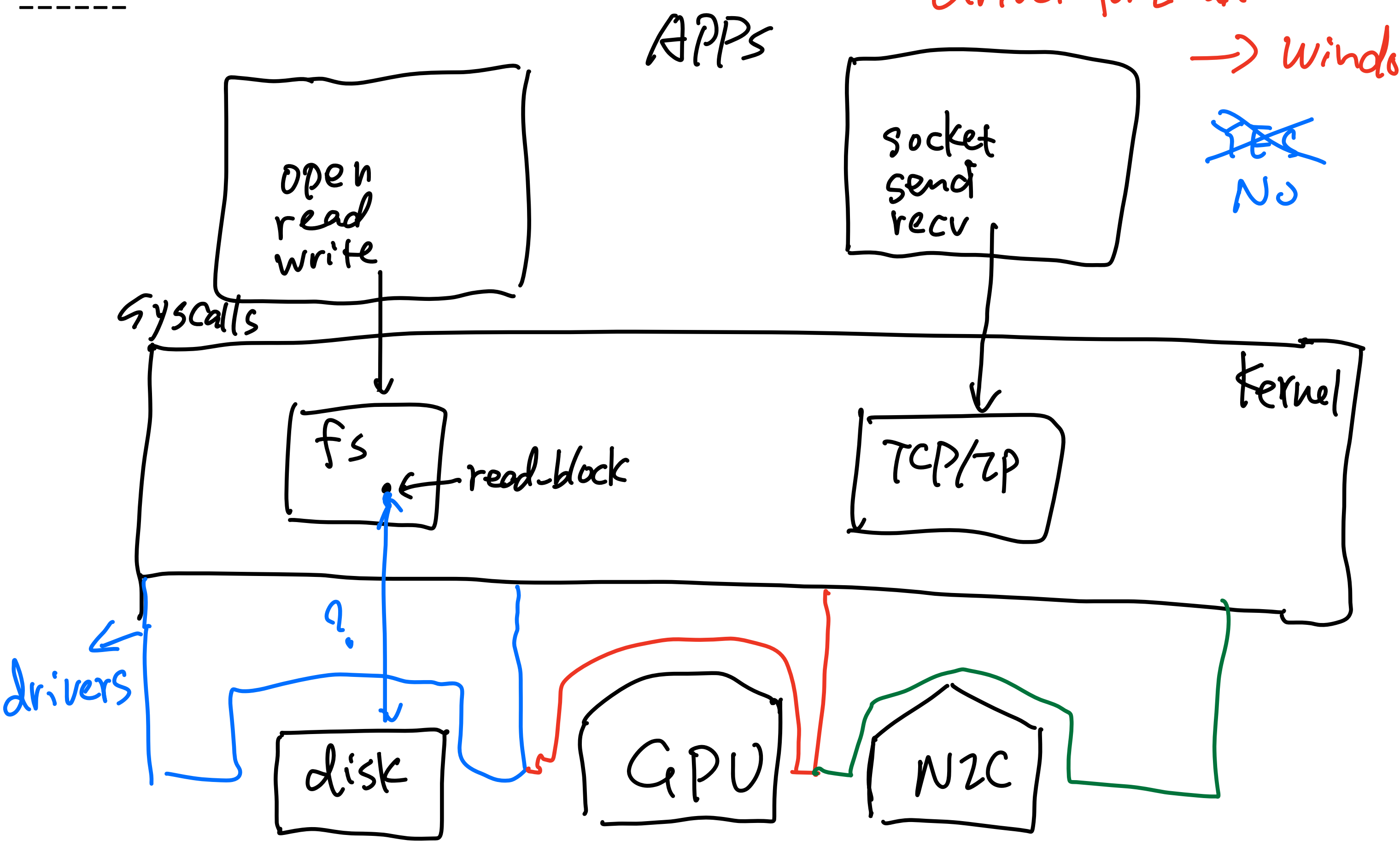
Week 13.b
CS3650
04/03 2024
https://naizhengtan.github.io/24spring/

1. Device drivers
2. Mechanics of communication
3. Demo: implementing a tty dev (egos-NU)
4. Communication configurations

Q: ① NVIDIA GPU driver → AMD GPU?

② NVIDIA GPU driver for Linux → windows?

~~YES~~ NO



2. Mechanics of communication between CPU and I/O devices

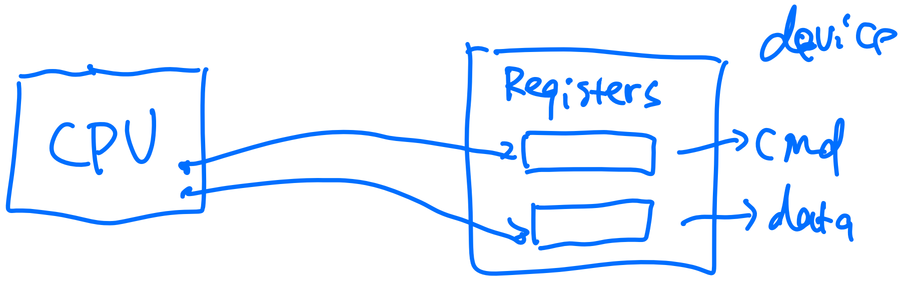
(a) explicit I/O instructions

(b) memory-mapped I/O

(c) interrupts

(d) through memory: both CPU and the device see the same memory
DMA

x86:
op: inb, outb, inw, outw
↳ 1B ↳ word 2B
operands: IO address



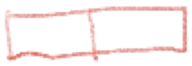
```
CS3650: select, synchronous I/O multiplexing

1. select interfaces

  a) select

    * int select(int nfds, fd_set *restrict readfds,
                 fd_set *restrict writefds, fd_set *restrict errorfds,
                 struct timeval *restrict timeout);

      select() examines the I/O descriptor sets whose addresses are passed in
      readfds, writefds, and errorfds to see if some of their descriptors are
      ready for reading, are ready for writing, or have an exceptional
      condition pending, respectively.

  b) fd_set manipulation

    * FD_ZERO(fd_set *set);            Clear all entries from the set.
    * FD_SET(int fd, fd_set *set);     Add fd to the set.
    * FD_CLR(int fd, fd_set *set);     Remove fd from the set.
    * FD_ISSET(int fd, fd_set *set);   Return true if fd is in the set.


2. An example - a chat server

    // Below is a code snippet using select()

    int fds[2] = {0, 0};
    fds[0] = ...                    // socket connection 1
    fds[1] = ...                    // socket connection 2

    fd_set readfds;

    while(1) {
        FD_ZERO(&readfds);
        for (int i=0; i<2; i++) {
            FD_SET(fds[i], &readfds);
        }

        int maxfd = ...             // Q: what is the maxfd?

        select(maxfd+1, &readfds, NULL, NULL, NULL);

        for (int i=0; i<2; i++) {
            if (FD_ISSET(fds[i], &readfds)) {
                print(fds[i], ...); // print msg received
            }
        }
    }
    ... // wrap up and exit
```

```
CS3650: I/O and device driver

1. An example of I/O instructions:
   Setting the cursor position

  The code below is excerpted from WeensyOS's k-hardware.c. It
  uses I/O instructions to set a blinking cursor in the upper left of
  the screen.

  // 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); // row
   outb(0x3D4, 15); // Command 15 = lower byte of position
   outb(0x3D5, cpos % 256); // column

  }
```

25

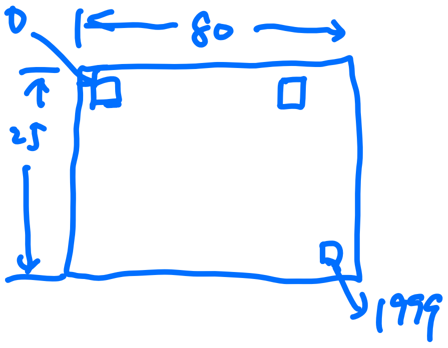
80

IO address



$80 \times 25 = 2000$

8bit ↔ $2^8=256$



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]

0x B8000

movq A, B

A address → read

B address → write

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

An example is in the console printing code from WeensyOS. Here is an excerpt from link/shared.ld:

```
/* Compare the address below to the map above. */
PROVIDE(console = 0xB8000);
```

+2000

This is an excerpt from lib.c; 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 "C" for "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;
  }
}
```

'e'

2000

?

'e'

2000

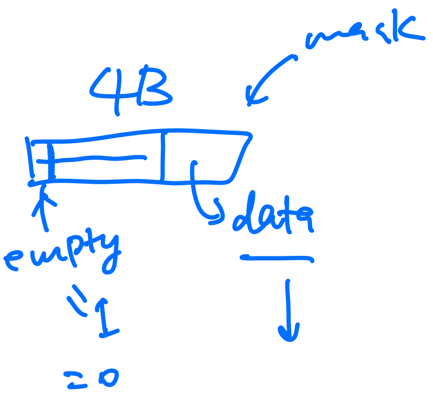
P

0xB8000
80
pos
he
25
80

Demo: How to implement a tty device (a terminal device using UART protocol)?

Base Address : 0x10013000

offset : 0x04 rxdata



- Configurations.
  - polling & interrupt

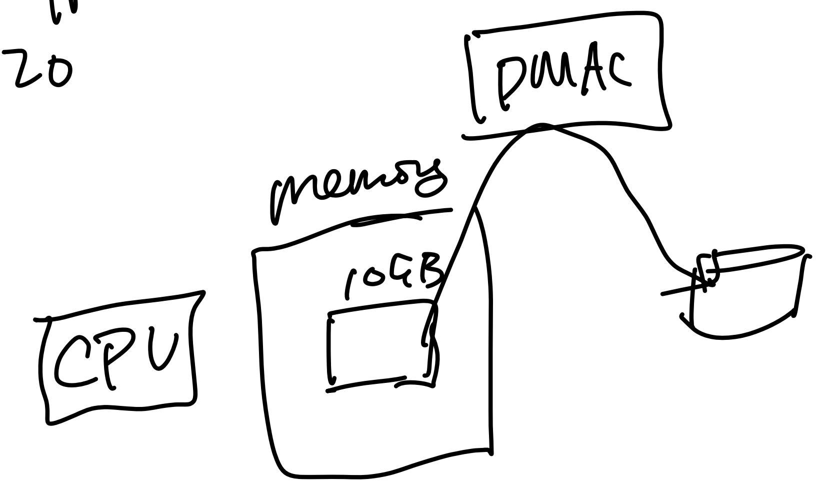
    interrupt x DMA

  - PIO & DMA

    IO address

    ↳ port-mapped IO

    ↳ MMIO



$\{$Polling, interrupt$\}$ X $\{$PIO, DMA$\}$