CS202 (-002): Lecture 16: I/O, maybe disks

Where we are

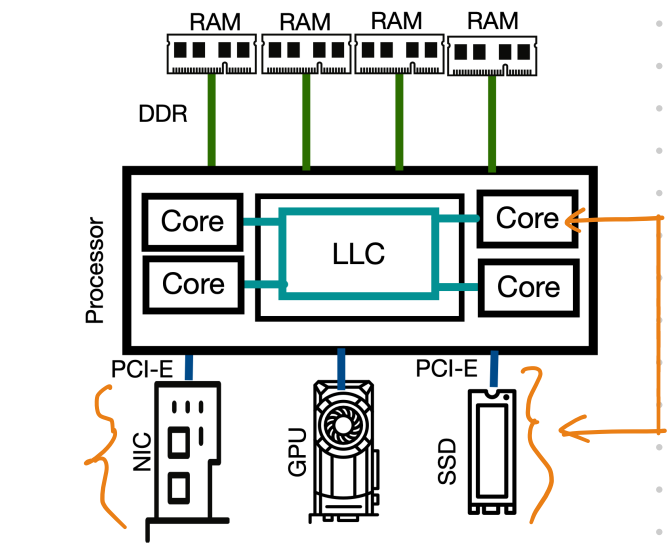
- Processes
  - → Allocate memory — Done, Virt memory
  - → Allocate CPU cores — Done, context switch (mechanism) Scheduling (policy)

What is left? Everything else on your computer

↳ UNIX philosophy: everything * is a file

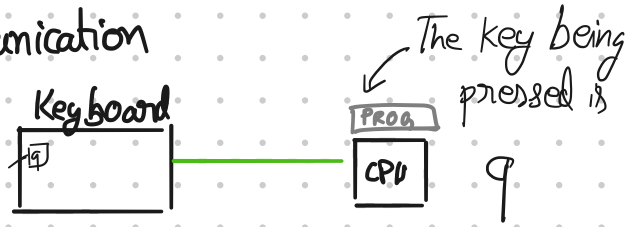
So look at how files & the file system work.

TODAY: The interface between programs (running on 1 or more CPU cores) & I/O Devices

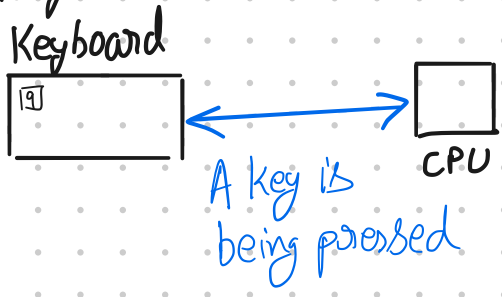


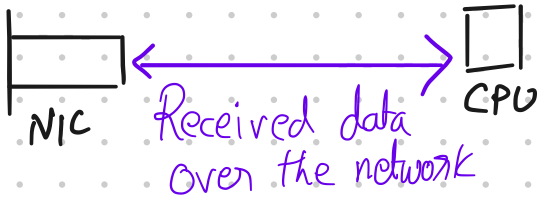
Need mechanisms for two things

ⓐ Communication



ⓑ Notification/Doorbell





Communication

Notification

(a) I/O instructions/ports

(b) MMIO

(c) DMA

(I) Interrupts

(II) Polling

...

I/O Ports

- in w/out w
  inb/outb

```
outb(0x1F2, 1);             // send 'count = 1' as an ATA argument
outb(0x1F3, src_sect);      // send 'src_sect', the sector number
outb(0x1F4, src_sect >> 8);
outb(0x1F5, src_sect >> 16);
outb(0x1F6, (src_sect >> 24) | 0xE0);
outb(0x1F7, 0x20);          // send the command: 0x20 = read sectors
```

Port

value

Port range	Summary
0x0000-0x001F	The first legacy DMA controller, often used for transfers to floppies.
0x0020-0x0021	The first Programmable Interrupt Controller
0x0022-0x0023	Access to the Model-Specific Registers of Cyrix processors.
0x0040-0x0047	The PIT (Programmable Interval Timer)
0x0060-0x0064	The "8042" PS/2 Controller or its predecessors, dealing with keyboards and mice.
0x0070-0x0071	The CMOS and RTC registers
0x0080-0x008F	The DMA (Page registers)
0x0092	The location of the fast A20 gate register
0x00A0-0x00A1	The second PIC
0x00C0-0x00DF	The second DMA controller, often used for soundblasters
0x00E9	Home of the Port E9 Hack. Used on some emulators to directly send text to the hosts' console.
0x0170-0x0177	The secondary ATA harddisk controller.
0x01F0-0x01F7	The primary ATA harddisk controller.
0x0278-0x027A	Parallel port
0x02F8-0x02FF	Second serial port
0x03B0-0x03DF	The range used for the IBM VGA, its direct predecessors, as well as any modern video card in legacy mode.
0x03F0-0x03F7	Floppy disk controller
0x03F8-0x03FF	First serial port

← In the olden days, but do not trust lists like this

Assume ↔ • Small amount (1 byte, 1 word (16-bits), 1 long (32-bits), etc.) of data need to be transferred

◦ Devices that every computer "ought to" have

Not so common/popular today.

Memory Mapped I/O (MMIO)

- You have seen this in Lab 4 (think console)

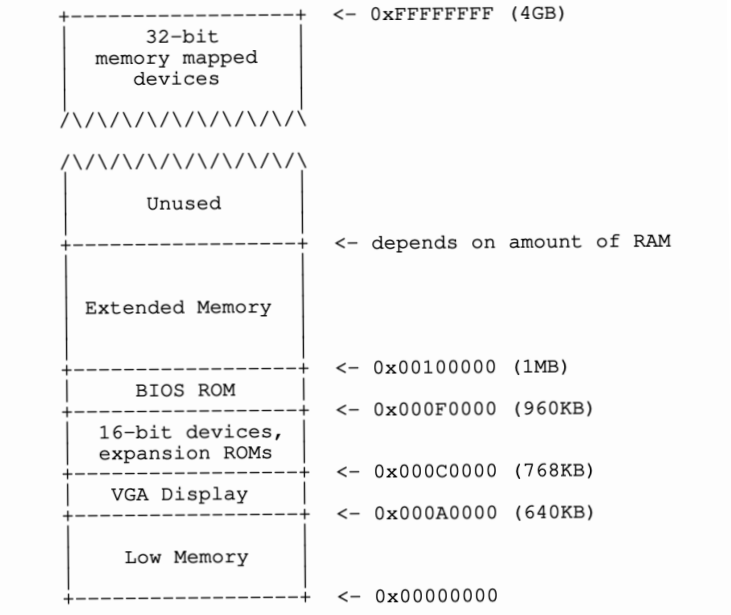
- Core idea:

Devices appear as physical memory

Reading/writing values to specific offsets

Dictated by the device

a. Here is a 32-bit PC's physical memory map:




```
00000000-00000fff : Reserved
00001000-0009ffff : System RAM
000a0000-000fffff : Reserved
  000a0000-000dffff : PCI Bus 0000:00
  000f0000-000fffff : System ROM
00100000-09e01fff : System RAM
09e02000-09ffffff : Reserved
0a000000-0a1fffff : System RAM
0a200000-0a20dfff : ACPI Non-volatile Storage
0a20e000-0affffff : System RAM
0b000000-0b01ffff : Reserved
0b020000-c7dd0fff : System RAM
c7dd1000-c7e2dfff : Reserved
c7e2e000-cb123fff : System RAM
cb124000-cb4a3fff : Reserved
  cb484000-cb487fff : MSFT0101:00
    cb484000-cb487fff : MSFT0101:00
  cb488000-cb48bfff : MSFT0101:00
    cb488000-cb48bfff : MSFT0101:00
cb4a4000-cb507fff : ACPI Tables
cb508000-ccc06fff : ACPI Non-volatile Storage
ccc07000-cdbfefff : Reserved
cdbff000-ceffffff : System RAM
cf000000-cfffffff : Reserved
d0000000-fec2ffff : PCI Bus 0000:00
  d0000000-e1ffffff : PCI Bus 0000:2b
    d0000000-dfffffff : 0000:2b:00.0
    e0000000-e1ffffff : 0000:2b:00.0
  f0000000-f7ffffff : PCI ECAM 0000 [bus 00-7f]
    f0000000-f7ffffff : pnp 00:00
  fb000000-fc0fffff : PCI Bus 0000:2b
    fb000000-fbffffff : 0000:2b:00.0
      fb000000-fbffffff : nvidia
    fc000000-fc07ffff : 0000:2b:00.0
    fc080000-fc083fff : 0000:2b:00.1
      fc080000-fc083fff : ICH HD audio
  fc200000-fc5fffff : PCI Bus 0000:02
    fc200000-fc4fffff : PCI Bus 0000:03
      fc200000-fc2fffff : PCI Bus 0000:2a
        fc200000-fc20ffff : 0000:2a:00.0
          fc200000-fc20ffff : r8169
        fc210000-fc213fff : 0000:2a:00.0
    fc300000-fc3fffff : PCI Bus 0000:05
      fc300000-fc303fff : 0000:05:00.0
      fc304000-fc304fff : 0000:05:00.0
```

You can look this up for your computer

Linux: cat /proc/iomem

```
link/shared.ld
2:PROVIDE(console = 0xB8000);
```

```
327     if (cpos < 0 || cpos >= CONSOLE_ROWS * CONSOLE_COLUMNS) {
328         cpos = 0;
329     }
330     cp.cursor = console + cpos;
```

```
309 static void console_putc(printer* p, unsigned char c, int color) {
310     console_printer* cp = (console_printer*) p;
311     if (cp->cursor >= console + CONSOLE_ROWS * CONSOLE_COLUMNS) {
312         cp->cursor = console;
313     }
314     if (c == '\n') {
315         int pos = (cp->cursor - console) % 80;
316         for (; pos != 80; pos++) {
317             *cp->cursor++ = ' ' | color;
318         }
319     } else {
320         *cp->cursor++ = c | color;
```

☆ 1) Communicate with device by reading & writing to

```
321     }
322 }
```

memory

② What to write & where determined by

device (& firmware)

③ Device ↔ core communication using memory requires coordination: Device & core have CONCURRENT access

Need to coordinate for safety

How? Defined by device.

Assumption ↔ Device exposes relatively small amount of state

↳ Not so true for

- Disks (several TBs)
- GPUs (almost as much or more than RAM)
- Net. cards (maybe)

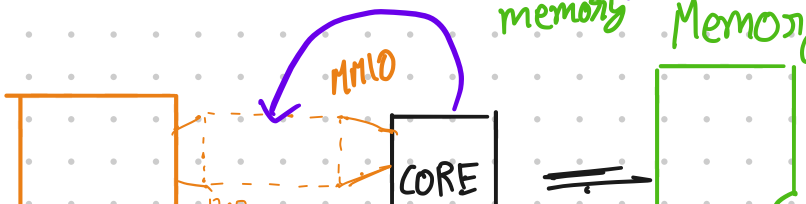
→ Much of the state does not need to be copied back-and-forth from memory

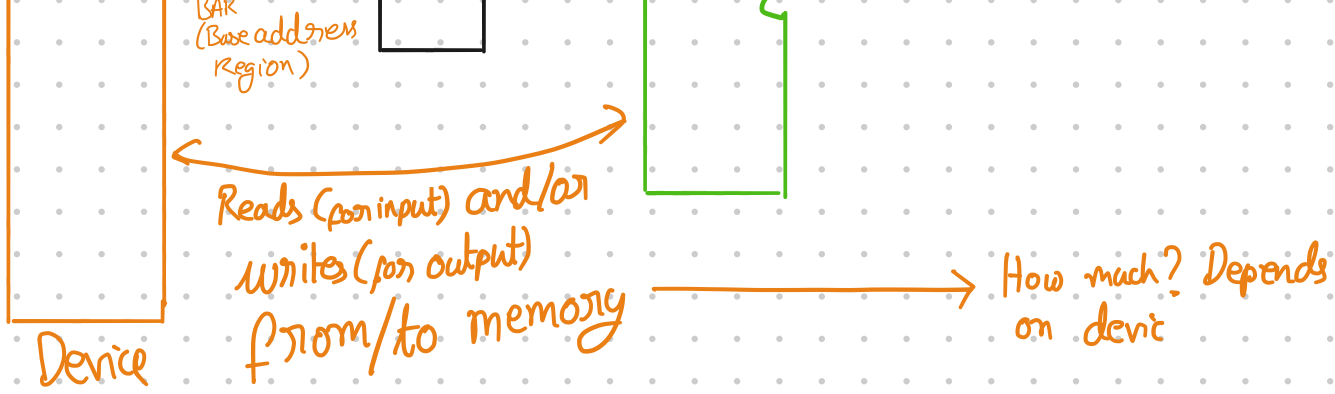
↳ Limited processing required for dev. output or input

Direct Memory Access (DMA)

Writes command, memory references ←→ Where & how

memory Memory (RAM) dictated by device





Also device specific

- Still needs coordination
  - ↳ Saye to read from memory ref. provided to device
  - → Saye to write
  - → Saye to provide next command

[This is what most devices use now]

Communication

- ✓ ⓐ I/O instructions/ports
- ✓ ⓑ MMIO
- ✓ ⓒ DMA

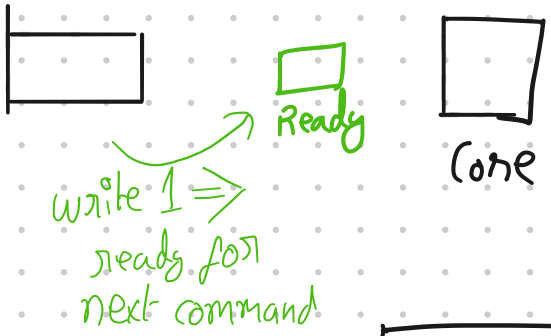
Notification

- Ⅰ Interrupts
- Ⅱ Polling
- ...

① Devices might not always have data available/be able to consume inputs

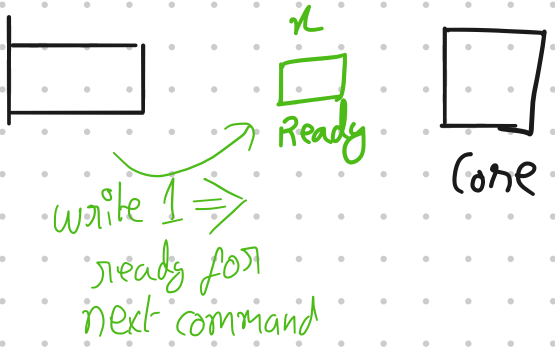
② As a part of coordination → might need to signal core to say condition is met [think spinlock →

mutex]



Notification

II Polling ⟷ Just have core repeatedly check location




```
while (x != 0) {
    ...
}
```

Spin to wait

```
84
85  // boot_waitdisk
86  //    Wait for the disk to be ready.
87  static void boot_waitdisk(void) {
88      // Wait until the ATA status register says ready (0x40 is on)
89      // & not busy (0x80 is off)
90      while ((inb(0x1F7) & 0xC0) != 0x40) {
91          /* do nothing */
92      }
93  }
94
```

+ Simpler code
+ Can have better perf. if notifications are common
− Uses CPU cycles.

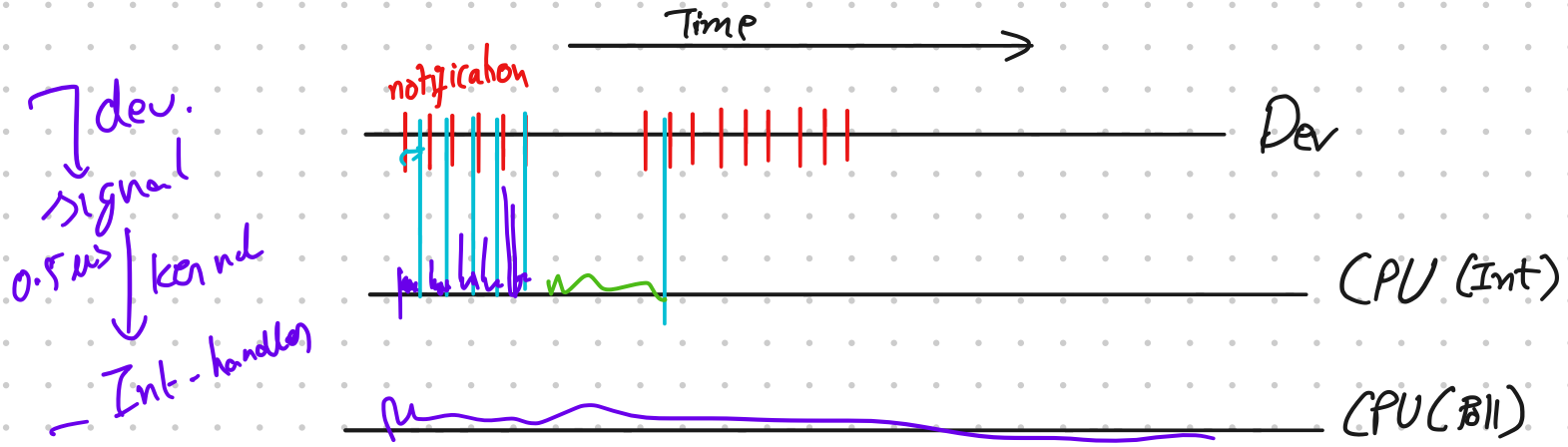
I Interrupt

⟶ Device sends a signal, triggering appropriate interrupt handler

Requires Registering with OS an int handler

+ Less CPU cycles

Interrupts vs Polling part

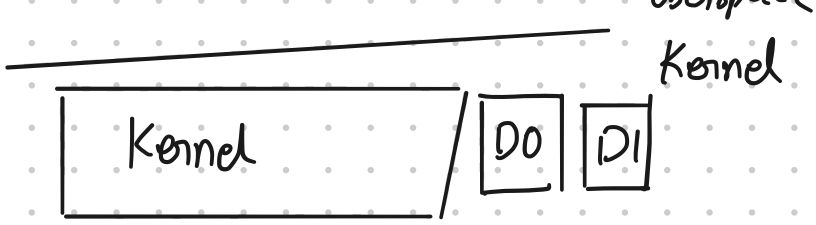


Drivers

- So far, answer to a lot of questions has been it depends on the device

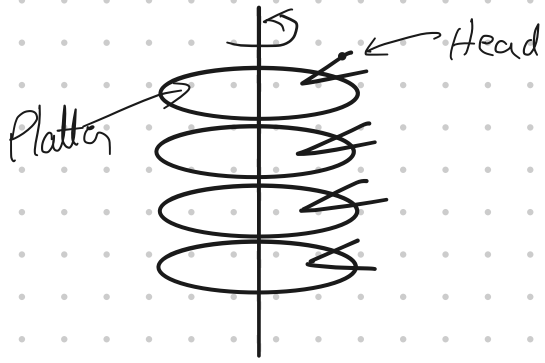
- But lots of devices, with new ones being added all the time

- (Usually) Support this by allowing the kernel to be extended
  - Drivers — support for specific device



Disks

- HDDs

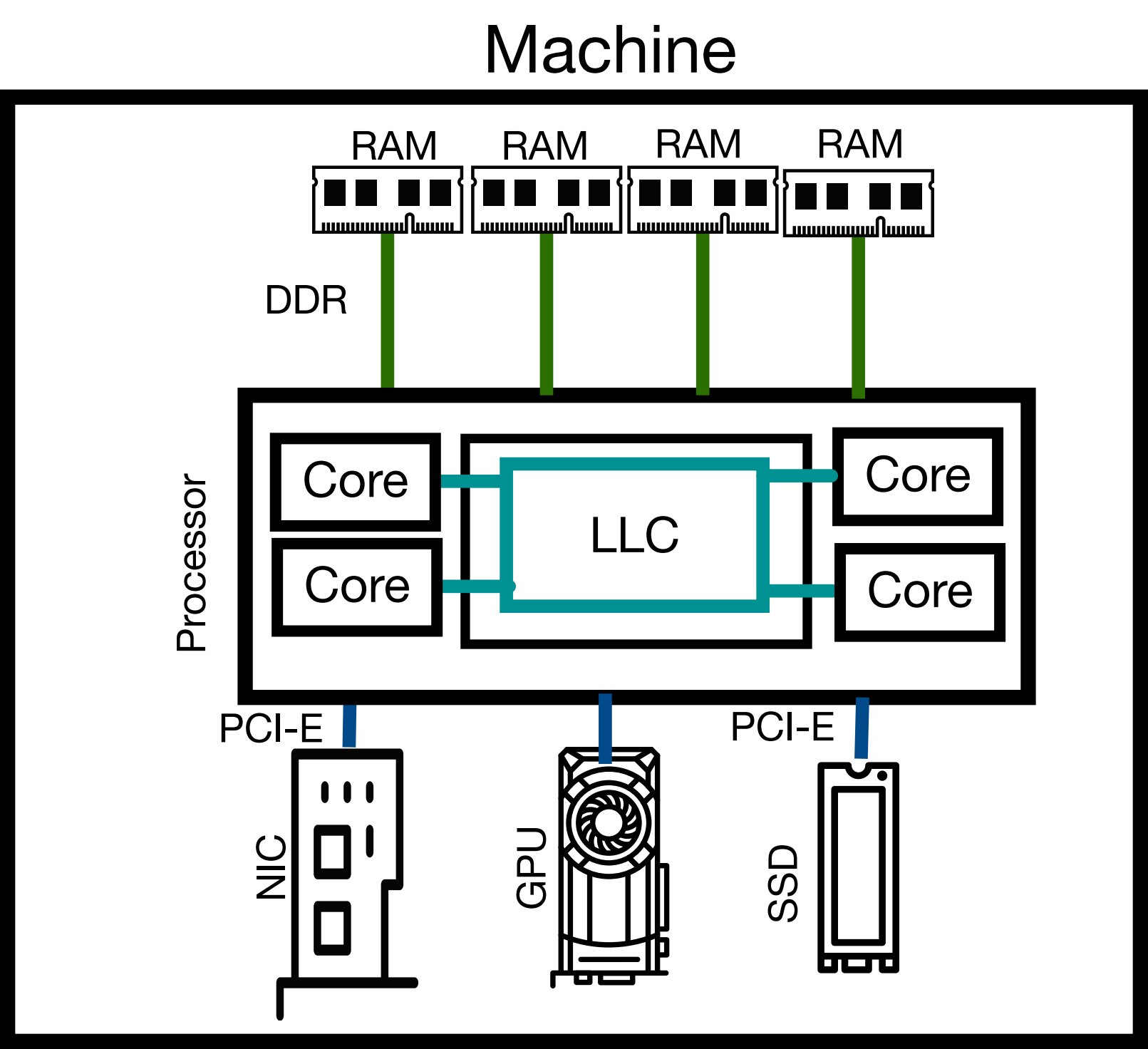


Time to read data

1. Position head (seek)
2. Wait to read data
   - ↳ Depends on location (maybe?)

Similarly for write.

But why HDDs?


Machine
RAM
RAM
RAM
RAM
DDR
Processor
Core
Core
LLC
Core
Core
PCI-E
PCI-E
NIC
GPU
SSD

```
CS 202, Fall 2023
Handout 9 (Class 17)

1. Example use of I/O instructions: boot loader

    Below is the WeensyOS boot loader

    It may be helpful to understand the overall picture

    This code demonstrates I/O, specifically with the disk: the
    bootloader reads in the kernel from the disk.

    See the functions boot_waitdisk() and boot_readsect(). Compare to Figures 36.5
    and 36.6 in OSTEP.

/* boot.c */
#include "x86-64.h"
#include "elf.h"

// boot.c
//
//   WeensyOS boot loader. Loads the kernel at address 0x40000 from
//   the first IDE hard disk.
//
//   A BOOT LOADER is a tiny program that loads an operating system into
//   memory. It has to be tiny because it can contain no more than 510 bytes
//   of instructions: it is stored in the disk's first 512-byte sector.
//
//   When the CPU boots it loads the BIOS into memory and executes it. The
//   BIOS intializes devices and CPU state, reads the first 512-byte sector of
//   the boot device (hard drive) into memory at address 0x7C00, and jumps to
//   that address.
//
//   The boot loader is contained in bootstart.S and boot.c. Control starts
//   in bootstart.S, which initializes the CPU and sets up a stack, then
//   transfers here. This code reads in the kernel image and calls the
//   kernel.
//
//   The main kernel is stored as an ELF executable image starting in the
//   disk's sector 1.

#define SECTORSIZE      512
#define ELFHDR          ((elf_header*) 0x10000) // scratch space

void boot(void) __attribute__((noreturn));
static void boot_readsect(uintptr_t dst, uint32_t src_sect);
static void boot_readseg(uintptr_t dst, uint32_t src_sect,
                         size_t filesz, size_t memsz);

// boot
//    Load the kernel and jump to it.
void boot(void) {
    // read 1st page off disk (should include programs as well as header)
    // and check validity
    boot_readseg((uintptr_t) ELFHDR, 1, PAGESIZE, PAGESIZE);
    while (ELFHDR->e_magic != ELF_MAGIC) {
        /* do nothing */
    }

    // load each program segment
    elf_program* ph = (elf_program*) ((uint8_t*) ELFHDR + ELFHDR->e_phoff);
    elf_program* eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ++ph) {
        boot_readseg(ph->p_va, ph->p_offset / SECTORSIZE + 1,
                     ph->p_filesz, ph->p_memsz);
    }

    // jump to the kernel
    typedef void (*kernel_entry_t)(void) __attribute__((noreturn));
    kernel_entry_t kernel_entry = (kernel_entry_t) ELFHDR->e_entry;
    kernel_entry();
}
```



```


// boot_readseg(dst, src_sect, filesz, memsz)
//    Load an ELF segment at virtual address `dst` from the IDE disk's sector
//    `src_sect`. Copies `filesz` bytes into memory at `dst` from sectors
//    `src_sect` and up, then clears memory in the range
//    `[dst+filesz, dst+memsz)`.
static void boot_readseg(uintptr_t ptr, uint32_t src_sect,
                         size_t filesz, size_t memsz) {
    uintptr_t end_ptr = ptr + filesz;
    memsz += ptr;

    // round down to sector boundary
    ptr &= ~(SECTORSIZE - 1);

    // read sectors
    for (; ptr < end_ptr; ptr += SECTORSIZE, ++src_sect) {
        boot_readsect(ptr, src_sect);
    }

    // clear bss segment
    for (; end_ptr < memsz; ++end_ptr) {
        *(uint8_t*) end_ptr = 0;
    }
}


// boot_waitdisk
//    Wait for the disk to be ready.
static void boot_waitdisk(void) {
    // Wait until the ATA status register says ready (0x40 is on)
    // & not busy (0x80 is off)
    while ((inb(0x1F7) & 0xC0) != 0x40) {
        /* do nothing */
    }
}


// boot_readsect(dst, src_sect)
//    Read disk sector number `src_sect` into address `dst`.
static void boot_readsect(uintptr_t dst, uint32_t src_sect) {
    // programmed I/O for "read sector"
    boot_waitdisk();
    outb(0x1F2, 1);             // send `count = 1` as an ATA argument
    outb(0x1F3, src_sect);      // send `src_sect`, the sector number
    outb(0x1F4, src_sect >> 8);
    outb(0x1F5, src_sect >> 16);
    outb(0x1F6, (src_sect >> 24) | 0xE0);
    outb(0x1F7, 0x20);          // send the command: 0x20 = read sectors

    // then move the data into memory
    boot_waitdisk();
    insl(0x1F0, (void*) dst, SECTORSIZE/4); // read 128 words from the disk
}


```

```
2. Two more examples of I/O instructions

    (a) Reading keyboard input

    The code below is an excerpt from WeensyOS's k-hardware.c

    This reads a character typed at the keyboard (which shows up on the
    "keyboard data port" (kEYBOARD_DATAREG)).

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

    int keyboard_readc(void) {
        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 ke
ys
            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;
    }
```



```
    (b) Setting the cursor position

    The code below is also excerpted from WeensyOS's k-hardware.c.  It
    uses I/O instructions to set a blinking cursor somewhere on a 25 x 80
    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);
        outb(0x3D4, 15);        // Command 15 = lower byte of position
        outb(0x3D5, cpos % 256);

    }
```

```
3. 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]


    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);

    /*
     * 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;
        }
    }
```