CS202

Last Class: Virtual memory — end

Context Switch

↳ WeenyOS

→ User Space

Today

- START ON FILESYSTEMS, DISKS, ETC. (Lab5)
  - ↳ mmap (2)
  - ↳ I/O MECHANISMS

mmap

Remember

↳ Demand paging

↳ Fetch pages into physical memory (from DISK) on access

→ virtual-memory-map

↳ Map virtual address ⟶ physical address

- Want to do both from userspace?

  ↳ mmap (2)

  [switch to Handout]

- Will implement in Lab 5

I/O

- The problem

↳ Most programs need to read or write to more than just memory

ⓐ Display (write)

ⓑ keyboard/mouse (read)

ⓒ Audio (read & write)

ⓓ ...

→ Instructions so far all about interacting with memory & registers

- mov/push/pop/...
- call/ret
- ...

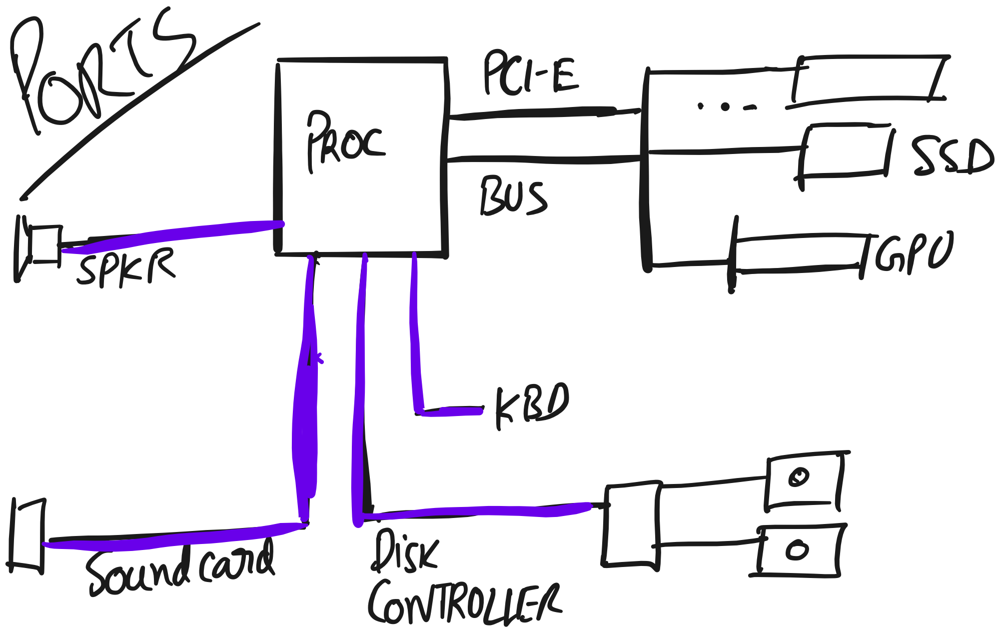
→ TODAY: Interacting with everything else

WARNING: It is messy (WHY? Historic baggage)

I/O ARCHITECTURE: GETTING DATA TO/FROM DEVICES

[Switch to Handout]

Slightly Abstracted (+ Old Details)



Explicit I/O Instructions

Context: Very old: processor knew all possible devices.

outb / outw : port, data

inb / inw : port, data

port.

Challenge : Figuring out [illegible]

Answer : Documentation + (Really /proc/ioports)

Memory Mapped I/O

Not All Physical Addresses Map To Memory

↳ Console In WeensyOS VGA Buffer

→ ...

Map Some Physical Addresses To Devices

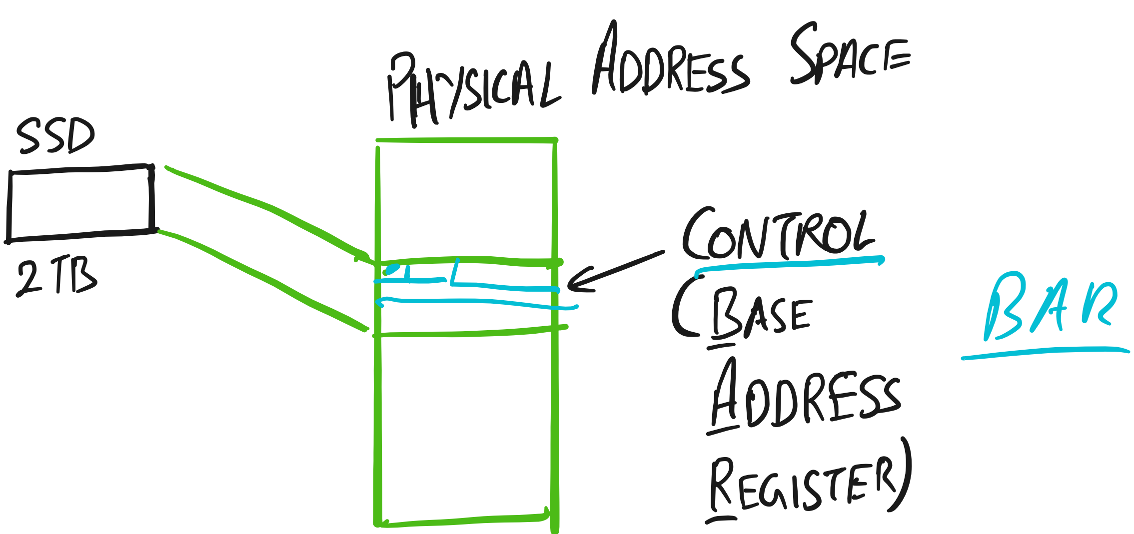
↓

Interact By Mapping To Virtual Address &

movq

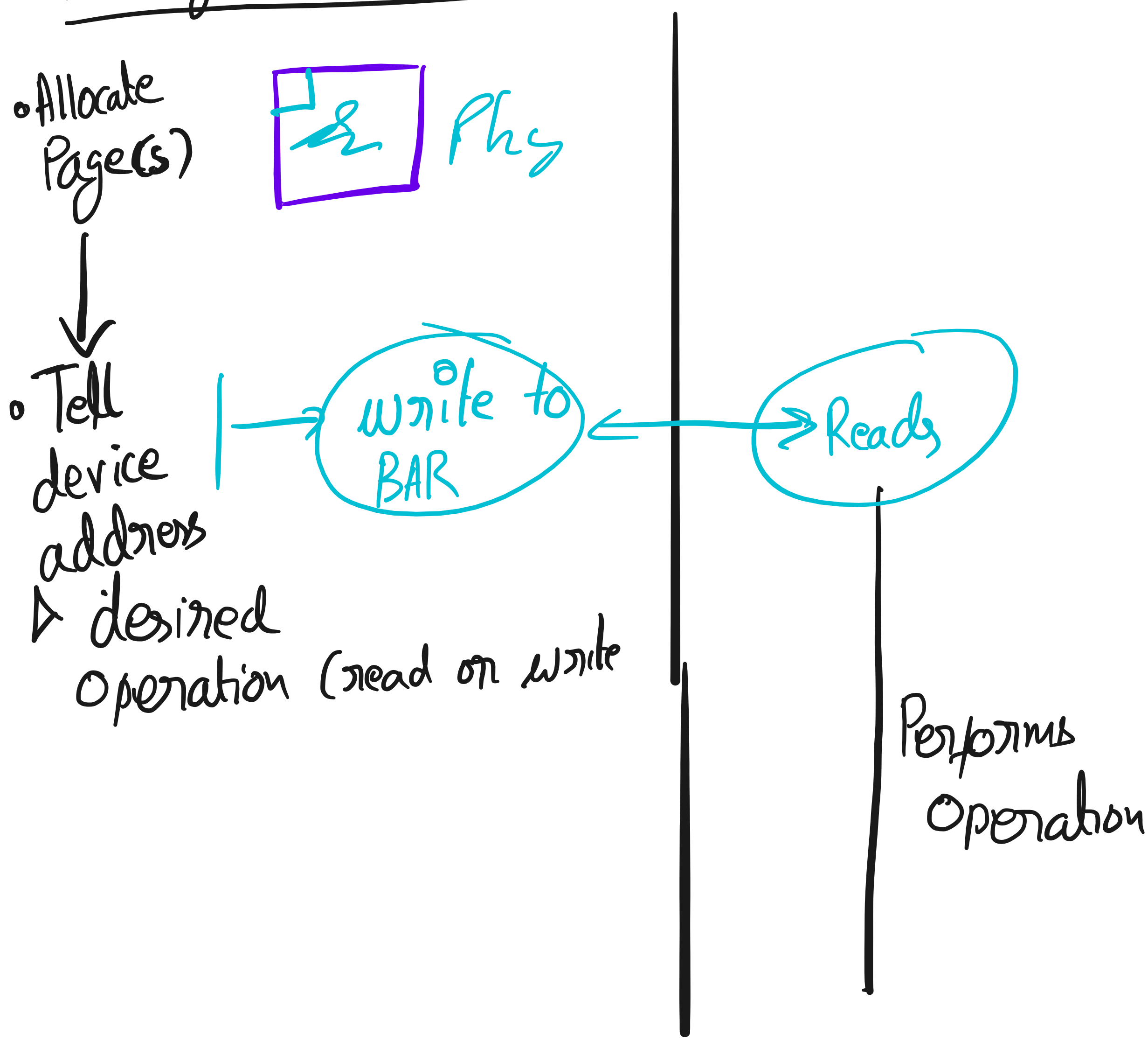
[Handout]

DMA



[illegible]

I/O PATH



Note: Must wait until completion before reading from page/freeing it

I/O Architecture: Coordination

MMIO: When is it safe for the program to overwrite an old value?

... to read value?

When is it safe to read data?

DMA: When is DMA finished

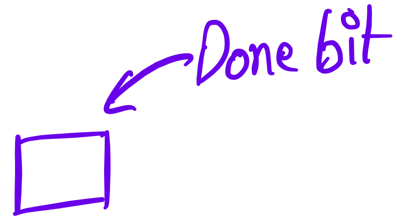
Broadly: How does device notify program that I/O is complete

Two Approaches

(a) Polling

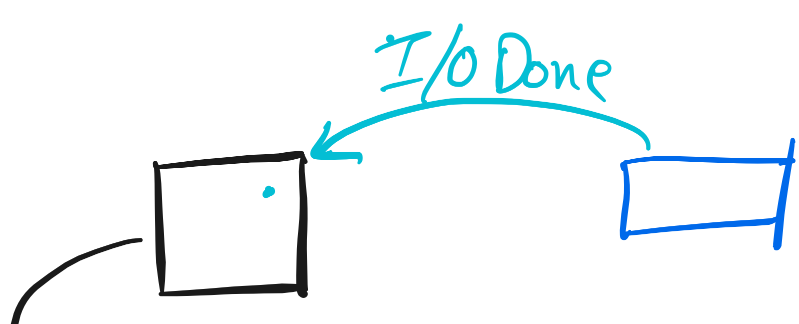
Software

```
while (done != 1){
    . wait
}
```



- Wastes cycles

(b) Interrupts



Interrupt handler

- Interrupts running application

Trade Off

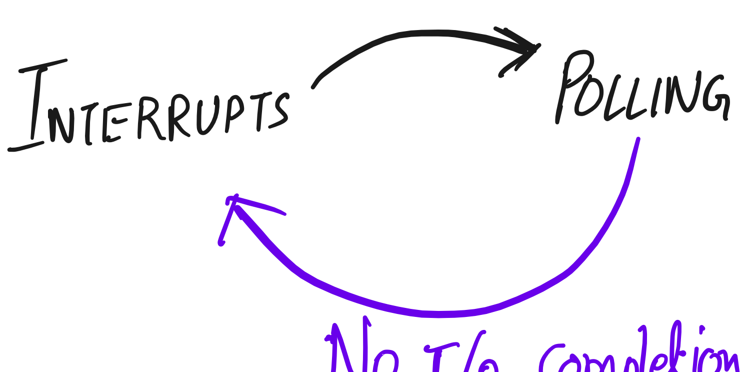
High Throughput device

↳ SSD (64GB/s ~ 16 million pages/s)

→ NIC (10-100 Gbps = 2.5-25 GB/s ~ 14.8 ~ 148 million packets /sec)

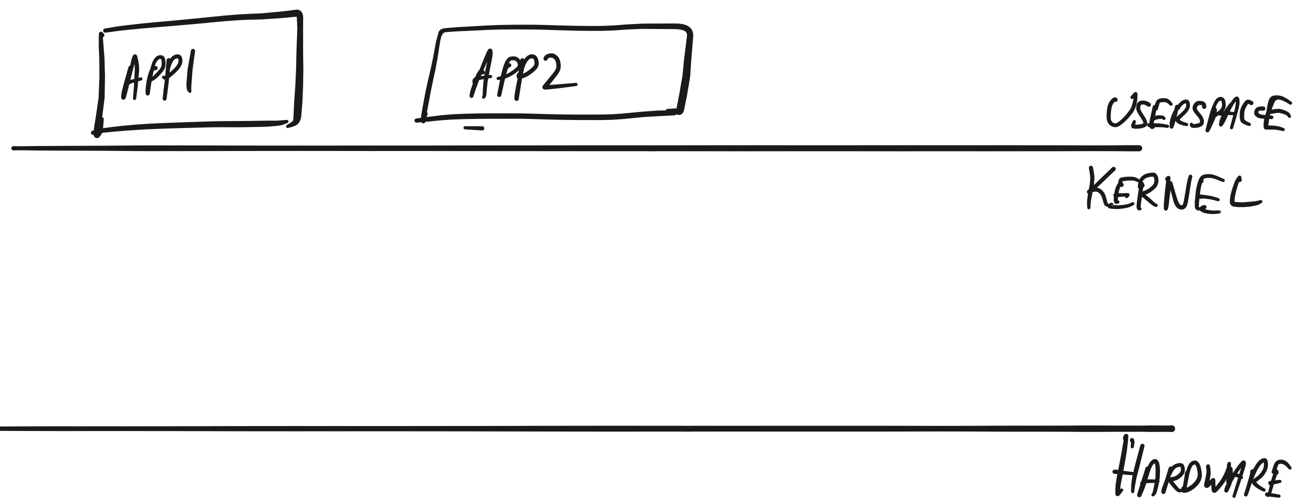
Interrupt per page/packet is too expensive

COMMON APPROACH



No I/O completed
for a while

Device Drivers

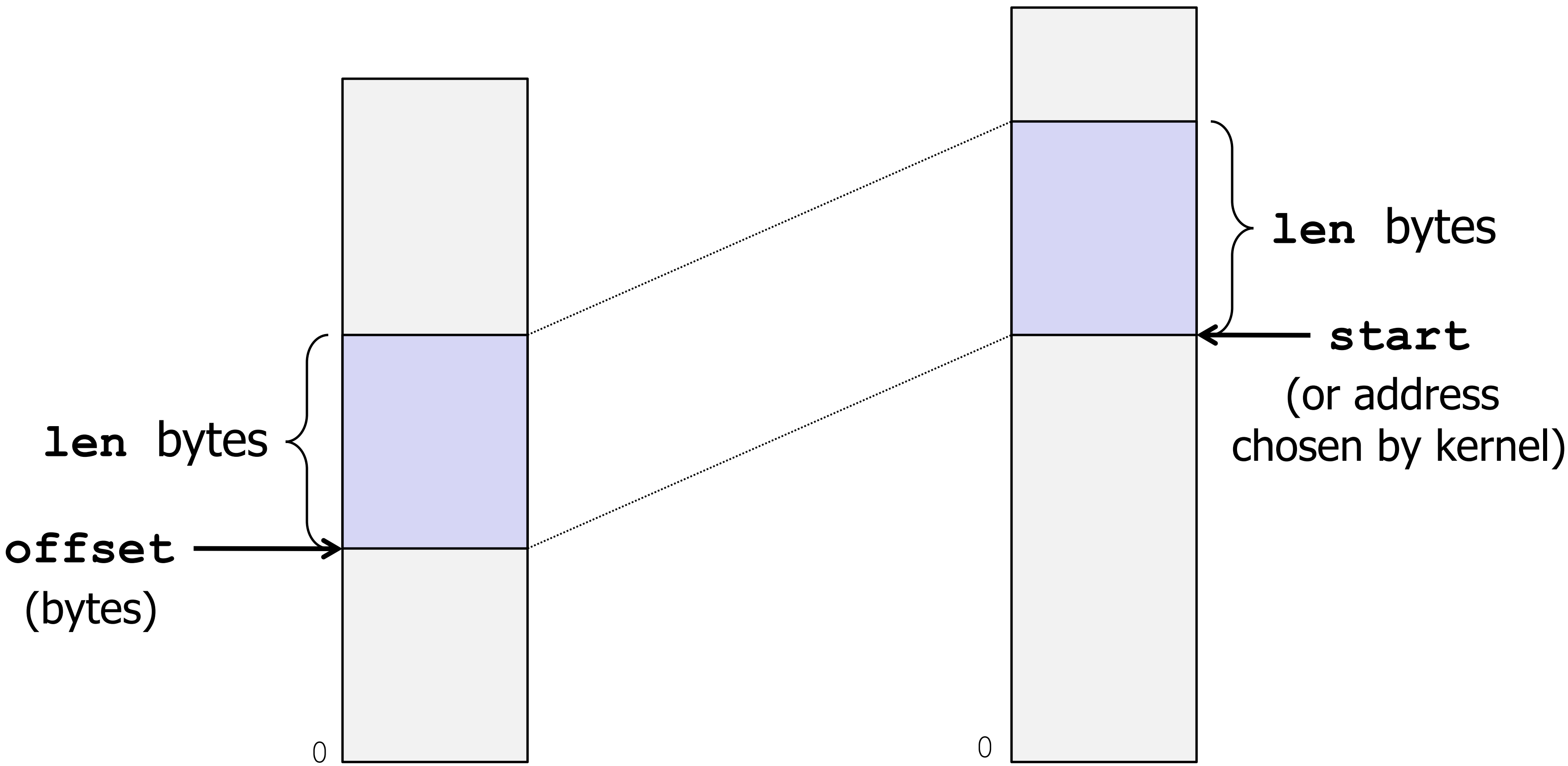


Blocking Vs Non-Blocking I/O

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

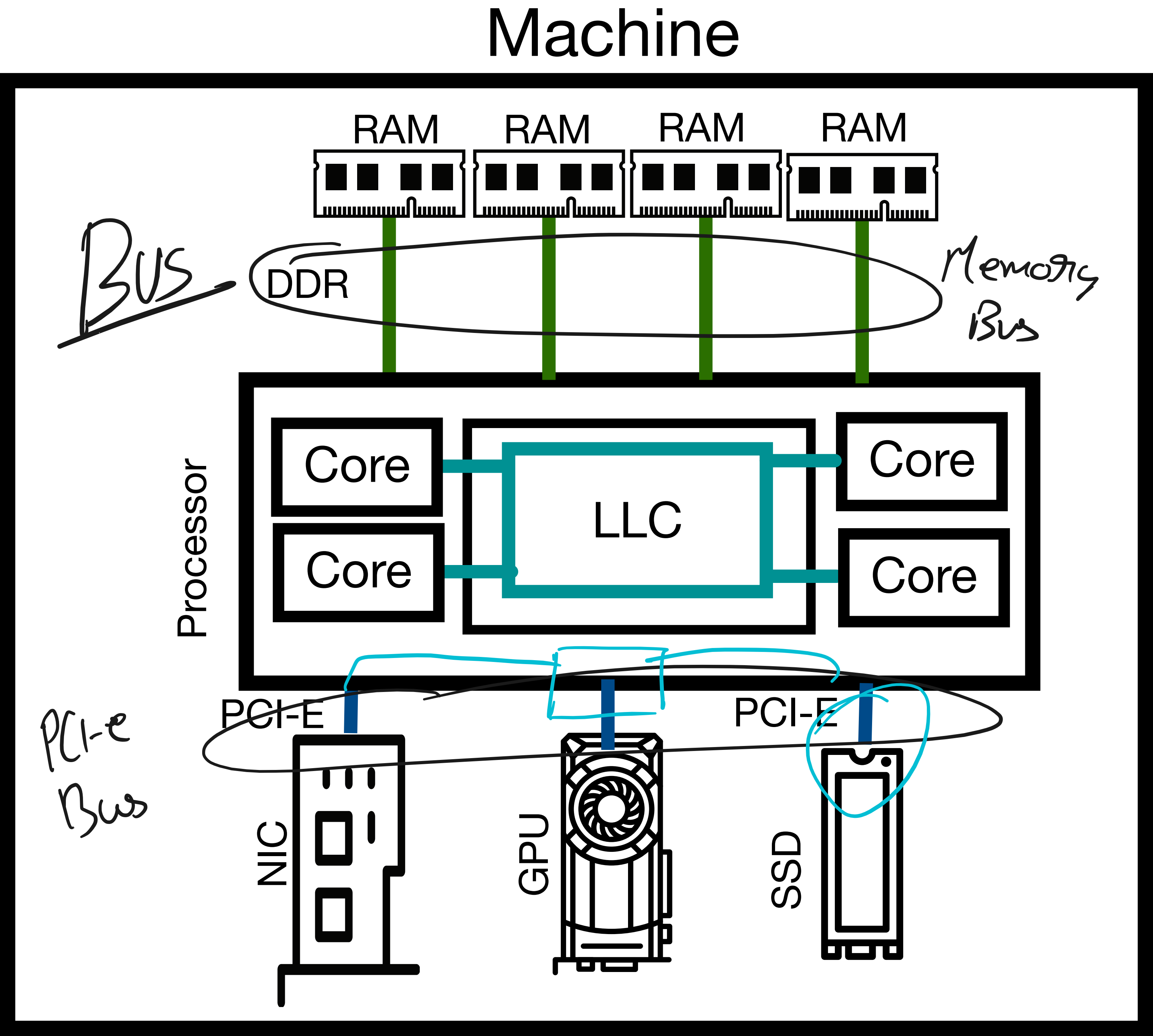
int fd = open('xx' ---)



Disk file specified by file descriptor **fd**

Process virtual memory

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

void mmapcopy(int fd, int size);

int main(int argc, char **argv) {
  struct stat stat;
  int fd;

  /* Check for required cmd line arg */
  if (argc != 2) {
    printf("usage: %s <filename>\n", argv[0]);
    exit(0);
  }

  /* Copy input file to stdout */
  if ((fd = open(argv[1], O_RDONLY, 0)) < 0)
    perror("open");

  fstat(fd, &stat);
  mmapcopy(fd, stat.st_size);

  close(fd);

  return 0;
}

void mmapcopy(int fd, int size) {

    /* Ptr to memory mapped area */
    char *bufp;

    bufp = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);

    write(STDOUT_FILENO, bufp, size);

    return;
}
```



Machine
RAM
RAM
RAM
RAM
Bus
DDR
Memory Bus
Processor
Core
Core
LLC
Core
Core
PCI-E
PCI-E
PCI-e Bus
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 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;
}
```


(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;
    }
}
```