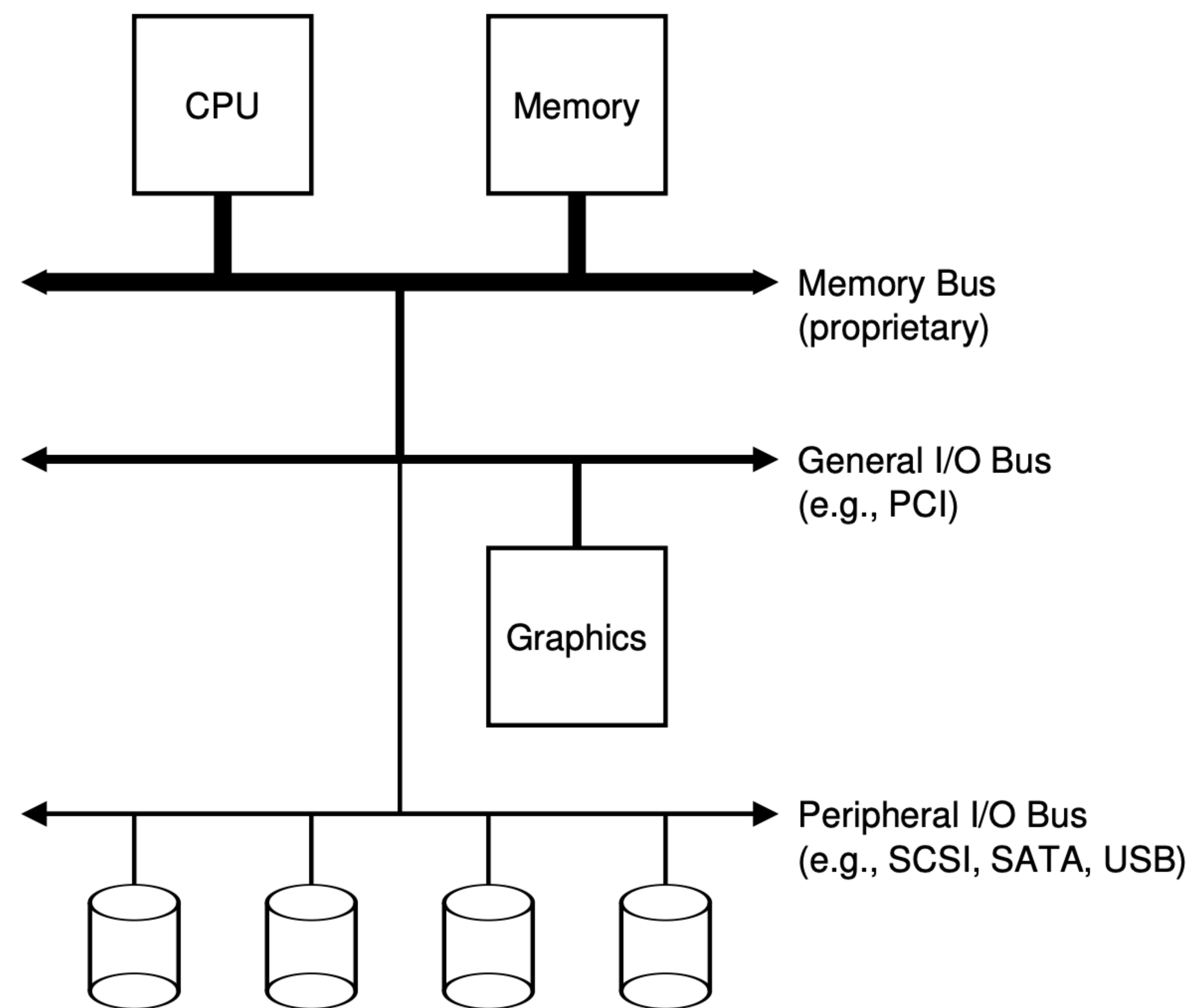
CS202 (003): Operating Systems
I/O

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish’s CS202

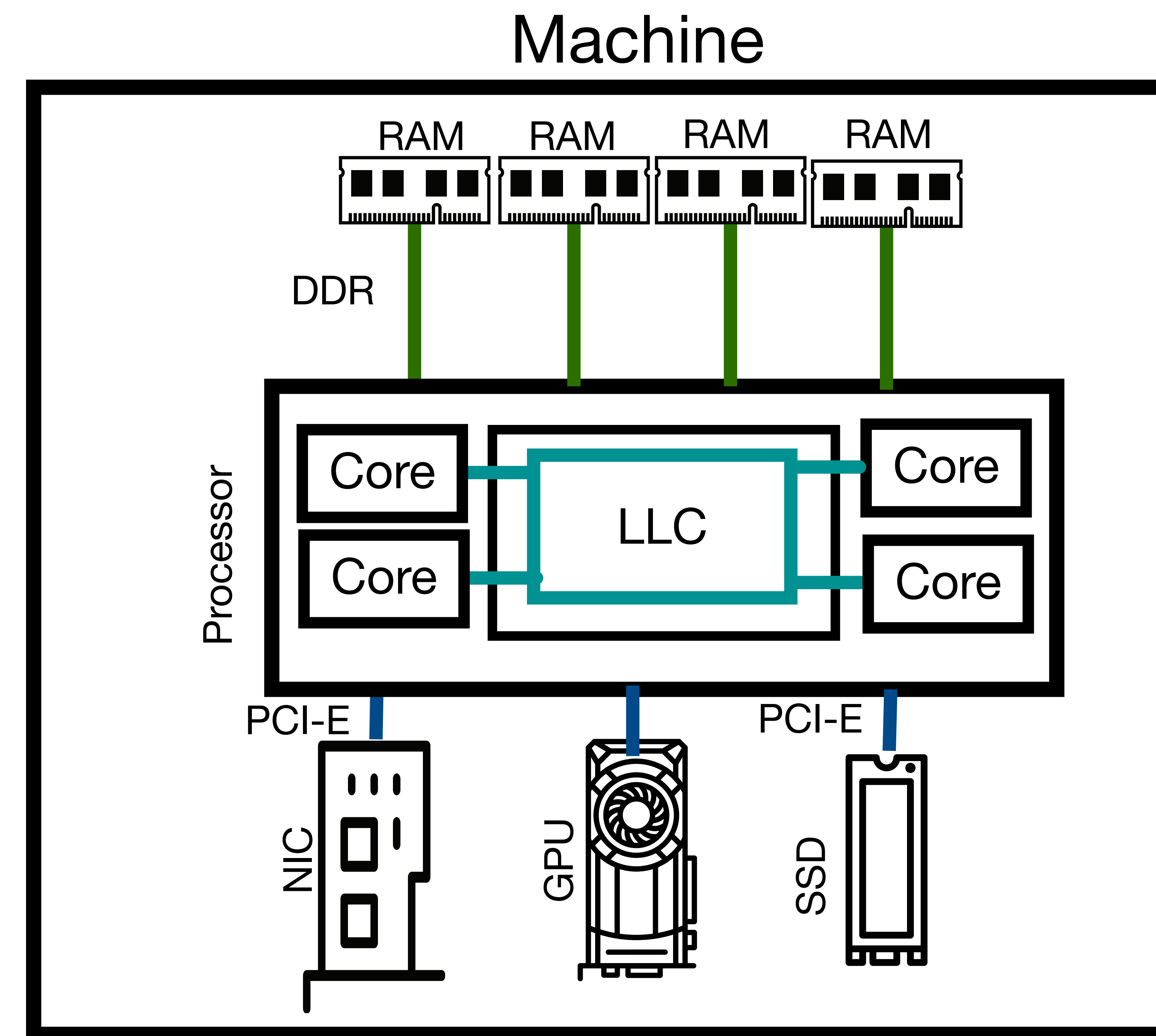
Last Time

I/O Architecture at a high-level



Textbook (older)

Figure 36.1: **Prototypical System Architecture**



Newer

CPU(kernel)/Device Interaction

Mechanics of Communication

Explicit I/O instructions `outb, inb, outw, inw` — WeesyOS boot.c (handout)

```
CS 202, Spring 2023
Handout 9 (Class 16)

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
}
```

```
Control Register:
  Address 0x3F6 = 0x08 (0000 1RE0): R=reset,
                  E=0 means "enable interrupt"

Command Block Registers:
  Address 0x1F0 = Data Port
  Address 0x1F1 = Error
  Address 0x1F2 = Sector Count
  Address 0x1F3 = LBA low byte
  Address 0x1F4 = LBA mid byte
  Address 0x1F5 = LBA hi  byte
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7      6     5     4    3    2    1     0
   BUSY  READY FAULT SEEK  DRQ  CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when ERROR==1)
    7      6     5     4    3    2    1    0
   BBK    UNC   MC   IDNF  MCR  ABRT T0NF AMNF

   BBK  = Bad Block
   UNC  = Uncorrectable data error
   MC   = Media Changed
   IDNF = ID mark Not Found
   MCR  = Media Change Requested
   ABRT = Command aborted
   T0NF = Track 0 Not Found
   AMNF = Address Mark Not Found
```

each register stores 8-bit data

Figure 36.5: **The IDE Interface**

(Integrated Device Electronics): connection between a bus on the motherboard and disk storage

LBA: Linear block address (28-bit in the handout)
describes storage locations
each sector gets a unique number starting from 0

Status is read-only
Command is used to add instructions to the disk
(e.g. 0xE0 is the write instruction)

https://www.rfcandy.biz/communication/ide.html

```
static int ide_wait_ready() {
  while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
    ;  // loop until drive isn't busy
  // return -1 on error, or 0 otherwise
}

static void ide_start_request(struct buf *b) {
  ide_wait_ready();
  outb(0x3f6, 0);                   // generate interrupt
  outb(0x1f2, 1);                   // how many sectors?
  outb(0x1f3, b->sector & 0xff);    // LBA goes here ...
  outb(0x1f4, (b->sector >> 8) & 0xff);   // ... and here
  outb(0x1f5, (b->sector >> 16) & 0xff);  // ... and here!
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);     // this is a WRITE
    outsl(0x1f0, b->data, 512/4);   // transfer data too!
  } else {
    outb(0x1f7, IDE_CMD_READ);      // this is a READ (no data)
  }
}

void ide_rw(struct buf *b) {
  acquire(&ide_lock);
  for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
    ;                               // walk queue
  *pp = b;                          // add request to end
  if (ide_queue == b)               // if q is empty
    ide_start_request(b);           // send req to disk
  while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
    sleep(b, &ide_lock);            // wait for completion
  release(&ide_lock);
}

void ide_intr() {
  struct buf *b;
  acquire(&ide_lock);
  if (!(b->flags & B_DIRTY) && ide_wait_ready() >= 0)
    insl(0x1f0, b->data, 512/4);    // if READ: get data
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);                        // wake waiting process
  if ((ide_queue = b->qnext) != 0) // start next request
    ide_start_request(ide_queue);   // (if one exists)
  release(&ide_lock);
}
```

handle read/write request

read: data is read and valid bit is set
write: data is written and dirty bit is cleaned

handle disk interrupt when operation complete

Figure 36.6: **The xv6 IDE Disk Driver (Simplified)**

CPU(kernel)/Device Interaction

Mechanics of Communication

Explicit I/O instructions

`outb, inb, outw, inw` — WeesyOS boot.c (handout)

`keyboard_readc()` — reading keyboard input (handout)

CPU(kernel)/Device Interaction

Mechanics of Communication

Explicit I/O instructions

`outb, inb, outw, inw` — WeesyOS boot.c (handout)

`keyboard_readc()` — reading keyboard input (handout)

`console_show_cursor()` — setting blinking cursor (handout)

```
129 2. Two more examples of I/O instructions
130
131     (a) Reading keyboard input
132
133     The code below is an excerpt from WeensyOS's k-hardware.c
134
135     This reads a character typed at the keyboard (which shows up on the
136     "keyboard data port" (kEYBOARD_DATAREG)).
137
138     /* Excerpt from WeensyOS x86-64.h */
139     // Keyboard programmed I/O
140     #define KEYBOARD_STATUSREG       0x64
141     #define KEYBOARD_STATUS_READY    0x01
142     #define KEYBOARD_DATAREG         0x60
143
144     int keyboard_readc(void) {
145         static uint8_t modifiers;
146         static uint8_t last_escape;
147
148         if ((inb(KEYBOARD_STATUSREG) & KEYBOARD_STATUS_READY) == 0) {
149             return -1;
150         }
151
152         uint8_t data = inb(KEYBOARD_DATAREG);
153         uint8_t escape = last_escape;
154         last_escape = 0;
155
156         if (data == 0xE0) {         // mode shift
157             last_escape = 0x80;
158             return 0;
159         } else if (data & 0x80) {   // key release: matters only for modifier ke
ys
160             int ch = keymap[(data & 0x7F) | escape];
161             if (ch >= KEY_SHIFT && ch < KEY_CAPSLOCK) {
162                 modifiers &= ~(1 << (ch - KEY_SHIFT));
163             }
164             return 0;
165         }
166
167         int ch = (unsigned char) keymap[data | escape];
168
169         if (ch >= 'a' && ch <= 'z') {
170             if (modifiers & MOD_CONTROL) {
171                 ch -= 0x60;
172             } else if (!(modifiers & MOD_SHIFT) != !(modifiers & MOD_CAPSLOCK))
{
173                 ch -= 0x20;
174             }
175         } else if (ch >= KEY_CAPSLOCK) {
176             modifiers ^= 1 << (ch - KEY_SHIFT);
177             ch = 0;
178         } else if (ch >= KEY_SHIFT) {
179             modifiers |= 1 << (ch - KEY_SHIFT);
180             ch = 0;
181         } else if (ch >= CKEY(0) && ch <= CKEY(21)) {
182             ch = complex_keymap[ch - CKEY(0)].map[modifiers & 3];
183         } else if (ch < 0x80 && (modifiers & MOD_CONTROL)) {
184             ch = 0;
185         }
186
187         return ch;
188     }
189
```

Handwritten notes on page 3:

- hardware ports / status flag for interact w/ keyboard.
- is keyboard ready to be read?
- read data.
- reset.
- signal for going into special key mode. (0xE0 + some other byte).
- map data to actual char using keymap.
- shift
- caps lock
- control key logic.
- special keys.
- function keys.


```
190
191     (b) Setting the cursor position
192
193     The code below is also excerpted from WeensyOS's k-hardware.c.  It
194     uses I/O instructions to set a blinking cursor somewhere on a 25 x 80
195     screen.
196
197     // console_show_cursor(cpos)
198     //    Move the console cursor to position 'cpos', which should be between 0
199     //    and 80 * 25.
200
201     void console_show_cursor(int cpos) {
202         if (cpos < 0 || cpos > CONSOLE_ROWS * CONSOLE_COLUMNS) {
203             cpos = 0;
204         }
205         outb(0x3D4, 14);        // Command 14 = upper byte of position
206         outb(0x3D5, cpos / 256);
207         outb(0x3D4, 15);        // Command 15 = lower byte of position
208         outb(0x3D5, cpos % 256);
209
210     }
211
212
213
214
```

Handwritten notes on page 4:

- VGA controller command port.
- VGA controller data port.
- split 10 bit # to 2 8 bit #.

CPU(kernel)/Device Interaction

Mechanics of Communication

Memory-mapped I/O

Most of the physical address space contains regular RAM
However, the lower memory addresses (650K-1MB) are special - they don't refer to actual RAM

WeesyOS console printing (handout)

```
1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/mman.h>
5  #include <sys/stat.h>
6  #include <sys/types.h>
7  #include <unistd.h>
8
9  void mmapcopy(int fd, int size);
10
11 int main(int argc, char **argv) {
12   struct stat stat;
13   int fd;
14
15   /* Check for required cmd line arg */
16   if (argc != 2) {
17     printf("usage: %s <filename>\n", argv[0]);
18     exit(0);
19   }
20
21   /* Copy input file to stdout */
22   if ((fd = open(argv[1], O_RDONLY, 0)) < 0)
23     perror("open");
24
25   fstat(fd, &stat);
26   mmapcopy(fd, stat.st_size);
27
28   close(fd);
29
30   return 0;
31 }
32
33 void mmapcopy(int fd, int size) {
34
35     /* Ptr to memory mapped area */
36     char *bufp;
37
38     bufp = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
39
40     write(STDOUT_FILENO, bufp, size);
41
42     return;
43 }
```

CPU$_{(kernel)}$/Device Interaction

Mechanics of Communication

Explicit I/O instructions

Memory-mapped I/O

Interrupts

CPU(kernel)/Device Interaction

Mechanics of Communication

Explicit I/O instructions

Memory-mapped I/O

Interrupts

Through memory

Both CPU and the device see the same memory, so they can use shared memory to communicate. (eg. DMA)

Polling vs. Interrupt (vs. busy waiting)

Busy Waiting

```
while (!device_is_ready()) {
  // CPU is stuck here,
repeatedly checking
  // Consuming CPU cycles doing
nothing useful
}
```

Simple but inefficient
CPU stuck here running at full speed

Polling vs. Interrupt (vs. busy waiting)

Busy Waiting

```
while (!device_is_ready()) {
  // CPU is stuck here,
repeatedly checking
  // Consuming CPU cycles doing
nothing useful
}
```

Polling

```
void poll_device() {
  while (1) {
     if (device_is_ready())
        { process_device_data(); }

      // Sleep/delay for a set
      interval
      sleep_ms(100); // Check every
      100ms
    }
  }
```

System checks device status at regular intervals

Lower CPU usage

Polling vs. Interrupt (vs. busy waiting)

Busy Waiting

```
while (!device_is_ready()) {
  // CPU is stuck here, repeatedly checking
  // Consuming CPU cycles doing nothing useful
}
```

Interrupts

```
// CPU sets up interrupt handler
set_interrupt_handler
   (device_interrupt_handler);

// CPU goes on to do other useful work
do_other_work();

// When device needs attention, it triggers interrupt
// and the handler runs automatically
void device_interrupt_handler() {
   // Handle the device's needs
   process_device_data();
}
```

Polling

```
void poll_device() {
   while (1) {
      if (device_is_ready())
         { process_device_data(); }

      // Sleep/delay for a set interval
      sleep_ms(100); // Check every 100ms
   }
}
```

Most sophisticated approach
If interrupt rate is high, we can get **livelock**

Programmed I/O vs. DMA (Direct memory access)

Programmed I/O

CPU writes data directly to device, and reads data directly from device.

Programmed I/O vs. DMA (Direct memory access)

Programmed I/O	DMA
CPU writes data directly to device, and reads data directly from device.	CPU places some buffers in main memory. Tells device where the buffers are Then "pokes" the device by writing to register Then device uses DMA to read or write the The CPU can poll to see if the DMA completed (or the device can interrupt the CPU when done). CPU don’t have to constantly deal with small amount of data transfer, the device can write the contents straight into memory

Software architecture: device drivers

Device drivers act as a bridge between hardware devices and the operating system kernel

Software architecture: device drivers

Device drivers act as a bridge between hardware devices and the operating system kernel

```
reset(): Initializes or resets the device
ioctl(): Provides device-specific controls
read()/write(): Standard data transfer operations
handle_interrupt(): Manages hardware interrupts
```

Software architecture: device drivers

Device drivers act as a bridge between hardware devices and the operating system kernel

```
reset(): Initializes or resets the device
ioctl(): Provides device-specific controls
read()/write(): Standard data transfer operations
handle_interrupt(): Manages hardware interrupts
```

Example interface

Advantages: don’t have to worry about the specific hardware implementation

Issues:

1. Device drivers is per-OS and per-device (hard part cannot be reused)
2. Bugs in device drivers often bring down the entire machine

Synchronous vs. Asynchronous I/O

Synchronous I/O

When a process makes a system call (like read() or write()),
it blocks (suspends) until the operation completes

The process enters a sleep state and cannot
execute other codes

Control returns to process after operation finishes

Code is often more readable, but it is slow

Synchronous vs. Asynchronous I/O

Synchronous I/O	Async I/O
When a process makes a system call (like read() or write()), it blocks (suspends) until the operation completes	System calls return immediately, even if the operation isn't completed
The process enters a sleep state and cannot execute other codes	Instead of blocking, the call returns a status indicating what would have happened
Control returns to process after operation finishes	Check through polling or interrupt

Need to use platform-specific extensions to POSIX to do async I/O for files