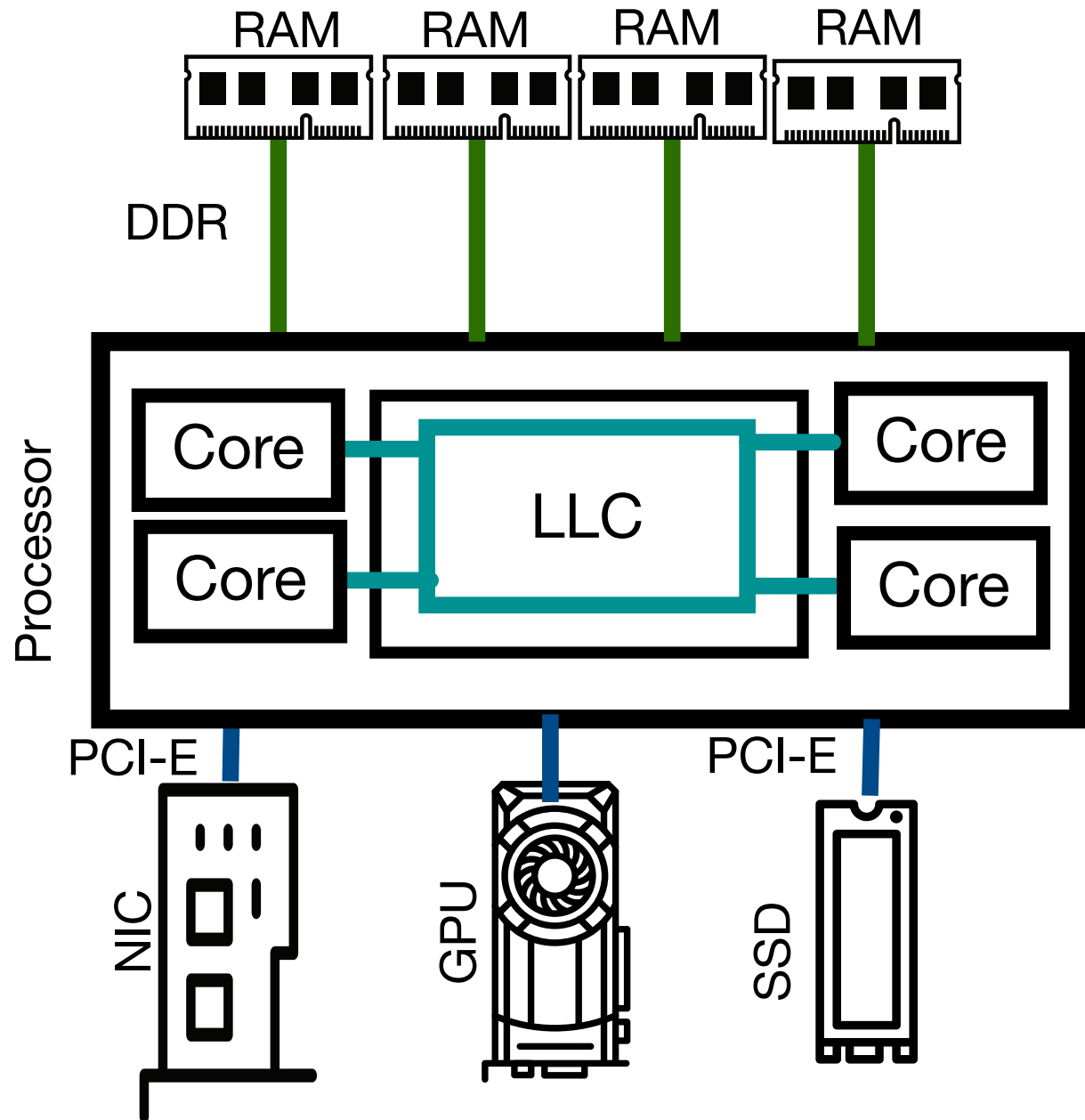


Machine



Mar 30, 25 13:24

handout11-2.txt

Page 1/5

```

1 CS 202
2 Handout 11 (Class 16)
3
4 1. Example use of I/O instructions: boot loader
5
6     Below is the WeensyOS boot loader
7
8     It may be helpful to understand the overall picture
9
10    This code demonstrates I/O, specifically with the disk: the
11    bootloader reads in the kernel from the disk.
12
13    See the functions boot_waitdisk() and boot_readsect(). Compare to Figures 36
14    and 36.6 in OSTEP.
15
16    /* boot.c */
17    #include "x86-64.h"
18    #include "elf.h"
19
20    // boot.c
21    //
22    // WeensyOS boot loader. Loads the kernel at address 0x40000 from
23    // the first IDE hard disk.
24    //
25    // A BOOT LOADER is a tiny program that loads an operating system into
26    // memory. It has to be tiny because it can contain no more than 510 bytes
27    // of instructions: it is stored in the disk's first 512-byte sector.
28    //
29    // When the CPU boots it loads the BIOS into memory and executes it. The
30    // BIOS initializes devices and CPU state, reads the first 512-byte sector of
31    // the boot device (hard drive) into memory at address 0x7C00, and jumps to
32    // that address.
33    //
34    // The boot loader is contained in bootstart.S and boot.c. Control starts
35    // in bootstart.S, which initializes the CPU and sets up a stack, then
36    // transfers here. This code reads in the kernel image and calls the
37    // kernel.
38    //
39    // The main kernel is stored as an ELF executable image starting in the
40    // disk's sector 1.
41
42    #define SECTORSIZE      512
43    #define ELFHDR          ((elf_header*) 0x10000) // scratch space
44
45    void boot(void) __attribute__((noreturn));
46    static void boot_readsect(uintptr_t dst, uint32_t src_sect);
47    static void boot_readseg(uintptr_t dst, uint32_t src_sect,
48                             size_t filesz, size_t memsz);
49
50    // boot
51    // Load the kernel and jump to it.
52    void boot(void) {
53        // read 1st page off disk (should include programs as well as header)
54        // and check validity
55        boot_readseg((uintptr_t) ELFHDR, 1, PAGE_SIZE, PAGE_SIZE);
56        while ((ELFHDR->e_magic != ELF_MAGIC) {
57            /* do nothing */
58        }
59
60        // load each program segment
61        elf_program* ph = (elf_program*) ((uint8_t*) ELFHDR + ELFHDR->e_phoff);
62        elf_program* eph = ph + ELFHDR->e_phnum;
63        for (; ph < eph; ++ph) {
64            boot_readseg(ph->p_va, ph->p_offset / SECTORSIZE + 1,
65                         ph->p_filesz, ph->p_memsz);
66        }
67
68        // jump to the kernel
69        typedef void (*kernel_entry_t)(void) __attribute__((noreturn));
70        kernel_entry_t kernel_entry = (kernel_entry_t) ELFHDR->e_entry;
71        kernel_entry();
72    }

```

Mar 30, 25 13:24

handout11-2.txt

Page 2/5

```

73
74
75    // boot_readseg(dst, src_sect, filesz, memsz)
76    // Load an ELF segment at virtual address 'dst' from the IDE disk's sector
77    // 'src_sect'. Copies 'filesz' bytes into memory at 'dst' from sectors
78    // 'src_sect' and up, then clears memory in the range
79    // '[dst+filesz, dst+memsz)'.
80    static void boot_readseg(uintptr_t ptr, uint32_t src_sect,
81                             size_t filesz, size_t memsz) {
82        uintptr_t end_ptr = ptr + filesz;
83        memsz += ptr;
84
85        // round down to sector boundary
86        ptr &= ~(SECTORSIZE - 1);
87
88        // read sectors
89        for (; ptr < end_ptr; ptr += SECTORSIZE, ++src_sect) {
90            boot_readsect(ptr, src_sect);
91        }
92
93        // clear bss segment
94        for (; end_ptr < memsz; ++end_ptr) {
95            *(uint8_t*) end_ptr = 0;
96        }
97    }
98
99
100    // boot_waitdisk
101    // Wait for the disk to be ready.
102    static void boot_waitdisk(void) {
103        // Wait until the ATA status register says ready (0x40 is on)
104        // & not busy (0x80 is off)
105        while ((inb(0x1F7) & 0xC0) != 0x40) {
106            /* do nothing */
107        }
108    }
109
110
111    // boot_readsect(dst, src_sect)
112    // Read disk sector number 'src_sect' into address 'dst'.
113    static void boot_readsect(uintptr_t dst, uint32_t src_sect) {
114        // programmed I/O for "read sector"
115        boot_waitdisk();
116        outb(0x1F2, 1); // send 'count = 1' as an ATA argument
117        outb(0x1F3, src_sect); // send 'src_sect', the sector number
118        outb(0x1F4, src_sect >> 8);
119        outb(0x1F5, src_sect >> 16);
120        outb(0x1F6, (src_sect >> 24) | 0xE0);
121        outb(0x1F7, 0x20); // send the command: 0x20 = read sectors
122
123        // then move the data into memory
124        boot_waitdisk();
125        insl(0x1F0, (void*) dst, SECTORSIZE/4); // read 128 words from the disk
126    }
127
128

```

Mar 30, 25 13:24

handout11-2.txt

Page 3/5

```

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
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             {
174                 ch -= 0x20;
175             }
176         } else if (ch >= KEY_CAPSLOCK) {
177             modifiers ^= 1 << (ch - KEY_SHIFT);
178             ch = 0;
179         } else if (ch >= KEY_SHIFT) {
180             modifiers |= 1 << (ch - KEY_SHIFT);
181             ch = 0;
182         } else if (ch >= CKEY(0) && ch <= CKEY(21)) {
183             ch = complex_keymap[ch - CKEY(0)].map[modifiers & 3];
184         } else if (ch < 0x80 && (modifiers & MOD_CONTROL)) {
185             ch = 0;
186         }
187         return ch;
188     }
189

```

Mar 30, 25 13:24

handout11-2.txt

Page 4/5

```

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 'cpo', which should be between 0
199     //     and 80 * 25.
200
201     void console_show_cursor(int cpos) {
202         if (cpo < 0 || cpo > CONSOLE_ROWS * CONSOLE_COLUMNS) {
203             cpo = 0;
204         }
205         outb(0x3D4, 14);          // Command 14 = upper byte of position
206         outb(0x3D5, cpo / 256);
207         outb(0x3D4, 15);          // Command 15 = lower byte of position
208         outb(0x3D5, cpo % 256);
209     }
210
211
212
213
214

```

Mar 30, 25 13:24

handout11-2.txt

Page 5/5

215 3. Memory-mapped I/O

216

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

218

```

219 +-----+ <- 0xFFFFFFFF (4GB)
220 | 32-bit |
221 | memory |
222 | mapped |
223 | devices|
224 +-----+
225 | \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ |
226 | \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ |
227 | |
228 | Unused |
229 | |
230 +-----+ <- depends on amount of RAM

```

231

```

232 | Extended Memory |
233 | |
234 +-----+
235 | BIOS ROM | <- 0x00100000 (1MB)
236 +-----+ <- 0x000F0000 (960KB)
237 | 16-bit devices, |
238 | expansion ROMs | <- 0x000C0000 (768KB)
239 +-----+ <- 0x000A0000 (640KB)
240 | VGA Display |
241 +-----+ <- 0x00000000
242 | Low Memory |
243 +-----+
244 | |
245 +-----+ <- 0x00000000
246
247

```

248

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

251

252

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

254

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

257

```

258 /* Compare the address below to the map above. */
259 PROVIDE(console = 0xB8000);
260
261

```

262

```

263 /*
264  * prints a character to the console at the specified
265  * cursor position in the specified color.
266  * Question: what is going on in the check
267  * if (c == '\n')
268  * ?
269  * Hint: '\n' is "C" for "newline" (the user pressed enter).
270  */
271
272 static void console_putc(printer* p, unsigned char c, int color) {
273     console_printer* cp = (console_printer*) p;
274     if (cp->cursor >= console + CONSOLE_ROWS * CONSOLE_COLUMNS) {
275         cp->cursor = console;
276     }
277     if (c == '\n') {
278         int pos = (cp->cursor - console) % 80;
279         for (; pos != 80; pos++) {
280             *cp->cursor++ = ' ' | color;
281         }
282     } else {
283         *cp->cursor++ = c | color;
284     }
285 }

```

286

287

288

289

290

291

292

293

294

295

296

297

298

299

300

301

302

303

304

305

306

307

308

309

310