



Code Listings

Listing A.1: irq.h

```
1  /*
2  Copyright (c) 2008-2015 Felix Freiling, University of Erlangen-Nürnberg, Germany
3  Copyright (c) 2011-2015 Hans-Georg Eßer, University of Erlangen-Nürnberg,
   Germany
4
5  This program is free software: you can redistribute it and/or modify it under
6  the terms of the GNU General Public License as published by the Free Software
7  Foundation, either version 3 of the License, or (at your option) any later
   version.
8
9  This program is distributed in the hope that it will be useful, but WITHOUT ANY
10 WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
11 PARTICULAR PURPOSE. See the GNU General Public License for more details.
12
13 You should have received a copy of the GNU General Public License along with
14 this program. If not, see <http://www.gnu.org/licenses/>. */
15
16 // this is irq.h
17
18
19 // IRQ numbers 0 to 7 are handled by the "master PIC", numbers 8 to 15 are
20 // handled by the "slave PIC".
21 // The slave PIC generates an IRQ number 2 on the master PIC.
22
23 #define IRQ_TIMER      0
24 #define IRQ_KBD        1
```

```

25 #define IRQ_SLAVE      2      // Here the slave PIC connects to master
26 #define IRQ_COM2       3
27 #define IRQ_COM1       4
28 #define IRQ_FDC         6
29 #define IRQ_IDE        14      // primary IDE controller; secondary has IRQ 15
30
31 // The two PICs (programmable interrupt controllers) must be configured:
32 // they have to be aware of their master/slave states and how the slave
33 // connects to the master.
34
35 // I/O Addresses of the PICs:
36
37 #define IO_PIC_MASTER_CMD 0x20 // Master (IRQs 0-7), command register
38 #define IO_PIC_MASTER_DATA 0x21 // Master, control register
39
40 #define IO_PIC_SLAVE_CMD 0xA0 // Slave (IRQs 8-15), command register
41 #define IO_PIC_SLAVE_DATA 0xA1 // Slave, control register
42
43 // When a PIC has raised an interrupt, it needs to be send an
44 // end of interrupt code; otherwise it will ignore further incoming
45 // interrupts.
46
47 #define END_OF_INTERRUPT 0x20
48
49 // Note that interrupts >7 will always involve both PICs, so both must
50 // be acknowledged with the end of interrupt code.
51
52
53
54 // Types
55
56 // idt_entry is an entry in the Interrupt Descriptor Table (IDT)
57
58 struct idt_entry {
59     unsigned int addr_low : 16; // lower 16 bits of address
60     unsigned int gdt_sel : 16; // use which GDT entry?
61     unsigned int zeroes : 8; // must be set to 0
62     unsigned int type : 4; // type of descriptor
63     unsigned int flags : 4;
64     unsigned int addr_high : 16; // higher 16 bits of address
65 } __attribute__((packed));
66
67
68 // idt_ptr describes address and size of the IDT. The address of
69 // this structure must be loaded in the IDTR (IDT register).
70
71 struct idt_ptr {
72     unsigned int limit : 16;
73     unsigned int base : 32;

```

```

74 } __attribute__((packed));
75
76
77
78 // Function prototypes
79
80 // This function is called during system initialization to setup the
81 // interrupt handling
82
83 void setup_irqs_and_faults ();

```

Listing A.2: irq.c

```

1  /*
2  Copyright (c) 2008-2015 Felix Freiling, University of Erlangen-Nürnberg, Germany
3  Copyright (c) 2011-2015 Hans-Georg Eßer, University of Erlangen-Nürnberg,
   Germany
4
5  This program is free software: you can redistribute it and/or modify it under
6  the terms of the GNU General Public License as published by the Free Software
7  Foundation, either version 3 of the License, or (at your option) any later
   version.
8
9  This program is distributed in the hope that it will be useful, but WITHOUT ANY
10 WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
11 PARTICULAR PURPOSE. See the GNU General Public License for more details.
12
13 You should have received a copy of the GNU General Public License along with
14 this program. If not, see <http://www.gnu.org/licenses/>. */
15
16
17 // this is irq.c
18 //
19 // This file contains C implementations of functions dealing with
20 // interrupt and fault management.
21 // Code for the (related) system call handler can be found in
22 // the syscall.c file.
23
24 #define __SRC_IRQ_C
25 #include "ulix.h"
26 #include "irq.h"
27 #include "tables.c"
28 #include "globals.h"
29
30
31 // The inport*() and outport*() functions retrieve a byte or word
32 // from a port or send it to the port.
33
34 byte
35 inportb (word port) {

```

```

36     byte retval;
37     asm volatile ("inb %%dx, %%al" : "=a"(retval) : "d"(port));
38     return retval;
39 }
40
41 word
42 inportw (word port) {
43     word retval;
44     asm volatile ("inw %%dx, %%ax" : "=a" (retval) : "d" (port));
45     return retval;
46 }
47
48 void
49 outportb (word port, byte data) {
50     asm volatile ("outb %%al, %%dx" : : "d" (port), "a" (data));
51 }
52
53 void
54 outportw (word port, word data) {
55     asm volatile ("outw %%ax, %%dx" : : "d" (port), "a" (data));
56 }
57
58
59 // fill_idt_entry() fills an entry in the interrupt descriptor table (IDT).
60 //
61 // arguments:
62 // - byte num: entry number (0..255)
63 // - unsigned long address: address of the interrupt handler function
64 // - word gdt sel: index into the global descriptor table (GDT), this
65 //   will always be set to 0x08 since we have prepared segment 0x08 as
66 //   code/kernel mode (see fill_gdt_entry in memory.c)
67 // - byte flags: flags for this selector; these will always be set to
68 //   0b1110 (1 = present, 11 = DPL 3, 0)
69 // - byte type: selector type; this will always be set to 0b1110
70 //   (32 bit interrupt gate)
71
72 void
73 fill_idt_entry (byte num, unsigned long address,
74                word gdt sel, byte flags, byte type) {
75     if (num >= 0 && num < 256) {
76         idt[num].addr_low = address & 0xFFFF; // address is the handler address
77         idt[num].addr_high = (address >> 16) & 0xFFFF;
78         idt[num].gdt sel = gdt sel; // GDT sel.: user or kernel mode?
79         idt[num].zeroes = 0;
80         idt[num].flags = flags;
81         idt[num].type = type;
82     }
83 }
84

```

```

85
86 // set_irqmask() sets the 16 bit IRQ mask.
87 //
88 // Since we use two PICs (with each one handling eight interrupts),
89 // we send the lower 8 bits to the primary PIC and the upper 8 bits
90 // to the secondary PIC.
91
92 void
93 set_irqmask (word mask) {
94     outportb (IO_PIC_MASTER_DATA, (char)(mask % 256) );
95     outportb (IO_PIC_SLAVE_DATA, (char)(mask >> 8) );
96 }
97
98
99 // get_irqmask() reads the IRQ mask.
100 //
101 // Similar to set_irqmask(), we need to query both PICs. They each
102 // return eight bits which we combine to form a 16 bit mask value.
103
104 word
105 get_irqmask () {
106     return inportb (IO_PIC_MASTER_DATA)
107         + (inportb (IO_PIC_SLAVE_DATA) << 8);
108 }
109
110
111 // enable_interrupt() enables a specific interrupt.
112 //
113 // It reads the IRQ mask, clears the bit indicated by number and
114 // writes the modified mask back to the PICs.
115
116 void
117 enable_interrupt (int number) {
118     set_irqmask (
119         get_irqmask ()           // the current value
120         & ~(1 << number)        // 16 one-bits, but bit "number" cleared
121     );
122 }
123
124
125 // FILL_IDT() calls fill_idt_entry() with default values. It is
126 // used for entering the fault0(), fault1(), ..., fault(31) and
127 // fault128() functions (which are defined in the assembler code).
128
129 #define FILL_IDT(i) \
130     fill_idt_entry (i, (unsigned int)fault##i, 0x08, 0b1110, 0b1110)
131
132
133 // setup_irqs_and_faults() initializes the PICs.

```

```

134 //
135 // - It calls the assembler function idt_load() to load the IDT
136 //   register (IDTR) of the CPU.
137 // - It then fills entries 0..31 and 128 of the IDT with the addresses
138 //   of the fault0(), ..., fault31() and fault128() fault handler
139 //   functions.
140 // - Then it programs the two PICs:
141 //   * It sets them up for cascade mode,
142 //   * remaps the the eight hardware interrupts from the primary
143 //     PIC to interrupt numbers 0x20..0x27,
144 //   * remaps the the eight hardware interrupts from the secondary
145 //     PIC to interrupt numbers 0x28..0x2f,
146 //   * enables 8086 mode
147 //   and initializes the IRQ mask (0xFFFF: all off).
148 // It enables the IRQ_SLAVE (2) line on the primary PIC so that it
149 // will act on interrupts received from the secondary PIC.
150 // - Finally it fills entries 32..47 of the IDT with the addresses
151 //   of the irq0(), ..., irq15() interrupt handler functions.
152
153 void
154 setup_irqs_and_faults () {
155     idtp.limit = (sizeof (struct idt_entry) * 256) - 1;    // must do -1
156     idtp.base = (int) &idt;
157     idt_load ();
158     FILL_IDT( 0); FILL_IDT( 1); FILL_IDT( 2); FILL_IDT( 3); FILL_IDT( 4);
159     FILL_IDT( 5); FILL_IDT( 6); FILL_IDT( 7); FILL_IDT( 8); FILL_IDT( 9);
160     FILL_IDT(10); FILL_IDT(11); FILL_IDT(12); FILL_IDT(13); FILL_IDT(14);
161     FILL_IDT(15); FILL_IDT(16); FILL_IDT(17); FILL_IDT(18); FILL_IDT(19);
162     FILL_IDT(20); FILL_IDT(21); FILL_IDT(22); FILL_IDT(23); FILL_IDT(24);
163     FILL_IDT(25); FILL_IDT(26); FILL_IDT(27); FILL_IDT(28); FILL_IDT(29);
164     FILL_IDT(30); FILL_IDT(31); FILL_IDT(128);
165     outportb (IO_PIC_MASTER_CMD, 0x11); // ICW1: initialize; begin programming
166     outportb (IO_PIC_SLAVE_CMD, 0x11); // ICW1: dito, for PIC2
167     outportb (IO_PIC_MASTER_DATA, 0x20); // ICW2 for PIC1: offset 0x20
168                                         // (remaps 0x00..0x07 -> 0x20..0x27)
169     outportb (IO_PIC_SLAVE_DATA, 0x28); // ICW2 for PIC2: offset 0x28
170                                         // (remaps 0x08..0x0f -> 0x28..0x2f)
171     outportb (IO_PIC_MASTER_DATA, 0x04); // ICW3 for PIC1: there's a slave on
172                                         // IRQ 2 (0b00000100 = 0x04)
173     outportb (IO_PIC_SLAVE_DATA, 0x02); // ICW3 for PIC2: your slave ID is 2
174     outportb (IO_PIC_MASTER_DATA, 0x01); // ICW4 for PIC1 and PIC2: 8086 mode
175     outportb (IO_PIC_SLAVE_DATA, 0x01);
176     // now the PICs are initialized and the remapping is complete
177     // next, we disable all interrupts except number 2 for the slave device
178     set_irqmask (0xFFFF); // initialize IRQ mask
179     enable_interrupt (IRQ_SLAVE); // IRQ slave
180
181     for (int i = 0; i < 16; i++) {
182         fill_idt_entry (32 + i,

```

```

183         (unsigned int)irqs[i],
184         0x08,
185         0b1110,    // flags: 1 (present), 11 (DPL 3), 0
186         0b1110);   // type: 1110 (32 bit interrupt gate)
187     }
188 }
189
190
191 // irq_handler() is the generic interrupt handler.
192 //
193 // It is called from the assembler functions irq0(), ..., irq15()
194 // which have already prepared the stack so that the interrupt number
195 // and process context are located on top of it (allowing irq_handler
196 // to access that data structure via its context_t *r argument).
197 //
198 // The primary PIC must be acknowledged via END_OF_INTERRUPT (in all
199 // cases). If the interrupt was raised by the secondary PIC, it must
200 // also be acknowledged. (Otherwise the PIC would stop sending
201 // further interrupt notifications.)
202 //
203 // If a handler function was entered in the interrupt_handlers[]
204 // table, it is called.
205
206 void
207 irq_handler (context_t *r) {
208     int number = r->int_no - 32;           // interrupt number
209     void (*handler)(context_t *r);        // type of handler functions
210
211     if (number >= 8) {
212         outportb (IO_PIC_SLAVE_CMD, END_OF_INTERRUPT); // notify slave PIC
213     }
214     outportb (IO_PIC_MASTER_CMD, END_OF_INTERRUPT);    // notify master PIC
215                                                         (always)
216
217     handler = interrupt_handlers[number];
218     if (handler != NULL) handler (r);
219 }
220
221 // install_interrupt_handler() installs a new interrupt handler
222 //
223 // arguments:
224 // - int irq:  entry number for the interrupt_handlers[] array (0..15)
225 // - void (*handler)(context_t *r):  address of the handler function
226 //
227 // If irq is a valid number, the given address is entered into the
228 // right entry of the array that stores the handler addresses.
229
230 void

```

```

231 install_interrupt_handler (int irq, void (*handler)(context_t *r)) {
232     if (irq >= 0 && irq < 16)
233         interrupt_handlers[irq] = handler;
234 }
235
236
237 // fault_handler() is the generic fault handler.
238 //
239 // The only fault that we can hope to handle satisfactorily is the
240 // page fault. Thus, fault_handler() starts by checking whether this
241 // fault was raised. If so, it executes the page_fault_handler() and
242 // returns.
243 //
244 // In case of a different fault, some debugging information is written
245 // to the terminal. Afterwards (if we are in user mode) the current
246 // process is terminated by calling syscall_exit(). This will invoke
247 // the scheduler which can pick a new process to continue execution.
248 //
249 // If the faulting address belongs to kernel code, we assume a major
250 // failure and jump to the kernel shell (which basically halts the
251 // system, as far as process execution is concerned; there is no
252 // reliable way back to user mode).
253
254 void
255 fault_handler (context_t *r) {
256     if (r->int_no == 14) {
257         // fault 14 is a page fault
258         page_fault_handler (r); return;
259     }
260
261     memaddress fault_address = (memaddress)(r->eip);
262
263     // interrupt number should be in 0..31
264     if (r->int_no < 32) {
265         // output debugging information
266         printf ("'%s' Exception at 0x%08x (task=%d, as=%d).\n",
267             exception_messages[r->int_no], r->eip, current_task, current_as);
268         printf ("eflags: 0x%08x  errcode: 0x%08x\n", r->eflags, r->err_code);
269         printf ("eax: %08x  ebx: %08x  ecx: %08x  edx: %08x\n",
270             r->eax, r->ebx, r->ecx, r->edx);
271         printf ("eip: %08x  esp: %08x  int: %8d  err: %8d\n",
272             r->eip, r->esp, r->int_no, r->err_code);
273         printf ("ebp: %08x  cs: 0x%02x  ds: 0x%02x  es: 0x%02x  "
274             "fs: 0x%02x  ss: 0x%02x\n",
275             r->ebp, r->cs, r->ds, r->es, r->fs, r->ss);
276         printf ("User mode stack: 0x%08x-0x%08x\n",
277             TOP_OF_USER_MODE_STACK - address_spaces[current_as].stacksize,
278             TOP_OF_USER_MODE_STACK);
279

```

```

280     if ( fault_address < 0xc0000000 ) {
281         // user mode: terminate current process
282         mutex_lock (thread_list_lock);
283         thread_table[current_task].state = TSTATE_ZOMBIE;
284         remove_from_ready_queue (current_task);
285         mutex_unlock (thread_list_lock);
286         r->ebx = -1; // exit_code for this process
287         syscall_exit (r);
288     }
289
290     // error inside the kernel, jump to kernel shell
291     scheduler_is_active = false; _set_statusline ("SCH:OFF", 16);
292     asm ("sti");
293     printf ("\n");
294     asm ("jmp kernel_shell");
295 }
296 }

```

Listing A.3: start.asm

```

1  ; Copyright (c) 2008-2015 Felix Freiling, University of Erlangen-Nürnberg,
   ; Germany
2  ; Copyright (c) 2011-2015 Hans-Georg Eßer, University of Erlangen-Nürnberg,
   ; Germany
3  ;
4  ; This program is free software: you can redistribute it and/or modify it under
5  ; the terms of the GNU General Public License as published by the Free Software
6  ; Foundation, either version 3 of the License, or (at your option) any later
7  ; version.
8  ;
9  ; This program is distributed in the hope that it will be useful, but WITHOUT
10 ; ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
11 ; FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
12 ;
13 ; You should have received a copy of the GNU General Public License along with
14 ; this program. If not, see <http://www.gnu.org/licenses/>.
15
16 ; this is ulix.c
17
18 bits 32
19
20 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
21
22 section .bss
23
24 global stack_first_address
25 global stack_last_address
26 stack_first_address:
27     resb 32*1024 ; reserve 32 KByte for the stack
28 stack_last_address:

```



```

78  section .text
79
80  higherhalf:
81      ; END higher half trick
82      mov     esp, _sys_stack ; set new stack
83      push    esp             ; save ESP
84      push    ebx             ; address of mboot structure (from GRUB)
85
86      extern  main             ; C function main() in ulix.c
87      call    main
88      jmp     $                ; infinite loop
89
90      extern  gp               ; defined in the C file
91
92  global gdt_flush
93  gdt_flush:
94      lgdt    [gp]
95      mov     ax, 0x10
96      mov     ds, ax
97      mov     es, ax
98      mov     fs, ax
99      mov     gs, ax
100     mov     ss, ax
101     jmp     0x08:flush2
102 flush2:
103     ret
104
105 global irq0, irq1, irq2, irq3, irq4, irq5, irq6, irq7
106 global irq8, irq9, irq10, irq11, irq12, irq13, irq14, irq15
107
108 %macro irq_macro 1
109     push     byte 0           ; error code (none)
110     push     byte %1          ; interrupt number
111     jmp     irq_common_stub ; rest is identical for all handlers
112 %endmacro
113
114 irq0:  irq_macro 32
115 irq1:  irq_macro 33
116 irq2:  irq_macro 34
117 irq3:  irq_macro 35
118 irq4:  irq_macro 36
119 irq5:  irq_macro 37
120 irq6:  irq_macro 38
121 irq7:  irq_macro 39
122 irq8:  irq_macro 40
123 irq9:  irq_macro 41
124 irq10: irq_macro 42
125 irq11: irq_macro 43
126 irq12: irq_macro 44

```

```

127 irq13: irq_macro 45
128 irq14: irq_macro 46
129 irq15: irq_macro 47
130
131 extern irq_handler           ; defined in the C source file
132 irq_common_stub:           ; this is the identical part
133     pusha
134     push    ds
135     push    es
136     push    fs
137     push    gs
138     push    esp             ; pointer to the context_t
139     call    irq_handler     ; call C function
140     pop     esp
141     pop     gs
142     pop     fs
143     pop     es
144     pop     ds
145     popa
146     add     esp, 8
147     iret
148
149 extern idtp                 ; defined in the C file
150 global idt_load
151 idt_load:
152     lidt [idtp]
153     ret
154
155 global fault0, fault1, fault2, fault3, fault4, fault5, fault6, fault7
156 global fault8, fault9, fault10, fault11, fault12, fault13, fault14, fault15
157 global fault16, fault17, fault18, fault19, fault20, fault21, fault22, fault23
158 global fault24, fault25, fault26, fault27, fault28, fault29, fault30, fault31
159
160 %macro fault_macro_0 1
161     push byte 0 ; error code
162     push byte %1
163     jmp fault_common_stub
164 %endmacro
165
166 %macro fault_macro_no0 1
167     ; don't push error code
168     push byte %1
169     jmp fault_common_stub
170 %endmacro
171
172 fault0: fault_macro_0      0    ; Divide by Zero
173 fault1: fault_macro_0      1    ; Debug
174 fault2: fault_macro_0      2    ; Non Maskable Interrupt
175 fault3: fault_macro_0      3    ; INT 3

```

```

176 fault4:  fault_macro_0      4  ;  INTO
177 fault5:  fault_macro_0      5  ;  Out of Bounds
178 fault6:  fault_macro_0      6  ;  Invalid Opcode
179 fault7:  fault_macro_0      7  ;  Coprocessor not available
180 fault8:  fault_macro_no0     8  ;  Double Fault
181 fault9:  fault_macro_0      9  ;  Coprocessor Segment Overrun
182 fault10: fault_macro_no0    10  ;  Bad TSS
183 fault11: fault_macro_no0    11  ;  Segment Not Present
184 fault12: fault_macro_no0    12  ;  Stack Fault
185 fault13: fault_macro_no0    13  ;  General Protection Fault
186 fault14: fault_macro_no0    14  ;  Page Fault
187 fault15: fault_macro_0      15  ;  (reserved)
188 fault16: fault_macro_0      16  ;  Floating Point
189 fault17: fault_macro_0      17  ;  Alignment Check
190 fault18: fault_macro_0      18  ;  Machine Check
191 fault19: fault_macro_0      19  ;  (reserved)
192 fault20: fault_macro_0      20  ;  (reserved)
193 fault21: fault_macro_0      21  ;  (reserved)
194 fault22: fault_macro_0      22  ;  (reserved)
195 fault23: fault_macro_0      23  ;  (reserved)
196 fault24: fault_macro_0      24  ;  (reserved)
197 fault25: fault_macro_0      25  ;  (reserved)
198 fault26: fault_macro_0      26  ;  (reserved)
199 fault27: fault_macro_0      27  ;  (reserved)
200 fault28: fault_macro_0      28  ;  (reserved)
201 fault29: fault_macro_0      29  ;  (reserved)
202 fault30: fault_macro_0      30  ;  (reserved)
203 fault31: fault_macro_0      31  ;  (reserved)
204
205 extern fault_handler
206 fault_common_stub:
207     pusha
208     push    ds
209     push    es
210     push    fs
211     push    gs
212     push    esp          ; pointer to the context_t
213     call    fault_handler ; call C function
214     pop     esp
215     pop     gs
216     pop     fs
217     pop     es
218     pop     ds
219     popa
220     add     esp, 8        ; for errcode, irq no.
221     iret
222
223 global tss_flush
224 tss_flush:

```

```

225     mov     ax, 0x28 | 0x03
226     ltr     ax                ; load the task register
227     ret
228
229     global cpu_usermode
230     cpu_usermode:
231         cli                ; disable interrupts
232         mov     ebp, esp      ; remember current stack address
233         mov     ax, 0x20 | 0x03 ; code selector 0x20 | RPL3: 0x03
234                                     ; RPL = requested protection level
235         mov     ds, ax
236         mov     es, ax
237         mov     fs, ax
238         mov     gs, ax
239         mov     eax, esp
240         push    0x20 | 0x03      ; code selector 0x20 | RPL3: 0x03
241         mov     eax, [ebp + 8]   ; stack address is 2nd argument
242         push    eax             ; stack pointer
243         pushf                ; EFLAGS
244         pop     eax             ; trick: reenale interrupts when doing iret
245         or      eax, 0x200
246         push    eax
247         push    0x18 | 0x03      ; code selector 0x18 | RPL3: 0x03
248         mov     eax, [ebp + 4]   ; return address (1st argument) for iret
249         push    eax
250         iret
251
252     extern syscall_handler
253     global fault128
254     fault128:
255         push    byte 0          ; put 128 on the stack so it looks the same
256         ; push    byte 128      ; as it does after a hardware interrupt
257         push    byte -128       ; (getting rid of nasm error for signed byte)
258         pusha
259         push    ds
260         push    es
261         push    fs
262         push    gs
263         push    esp ; pointer to the context_t
264         call    syscall_handler
265         pop     esp
266         pop     gs
267         pop     fs
268         pop     es
269         pop     ds
270         popa
271         add     esp, 8           ; undo the two "push byte" commands from the start_
272         iret
273

```

```

274 global get_eip
275 get_eip:
276     pop     eax        ; top of stack contains return address
277     push    eax        ; write it back
278     ret

```

Listing A.4: ulix.c

```

1  /*
2  Copyright (c) 2008-2014 Felix Freiling, University of Erlangen-Nürnberg, Germany
3  Copyright (c) 2011-2014 Hans-Georg Eßer, University of Erlangen-Nürnberg,
   Germany
4
5  This program is free software: you can redistribute it and/or modify it under
6  the terms of the GNU General Public License as published by the Free Software
7  Foundation, either version 3 of the License, or (at your option) any later
   version.
8
9  This program is distributed in the hope that it will be useful, but WITHOUT ANY
10 WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
11 PARTICULAR PURPOSE. See the GNU General Public License for more details.
12
13 You should have received a copy of the GNU General Public License along with
14 this program. If not, see <http://www.gnu.org/licenses/>. */
15
16 // this is ulix.c
17 #define __SRC_ULIX_C
18 #include "ulix.h"
19 #include "irq.h"
20
21 // global variables
22 #include "globals.h"
23 #include "tables.c"
24
25 // other source files
26 #include "kshell.c" // kernel_shell()
27
28
29 // kernel main() function
30
31 void main () {
32     initialize_all_locks (); // locks and semaphores
33     initialize_all_queues (); // ready and standard blocked queues
34     uartinit (1); // serial port (debugging)
35     initialize_memory (); // paging, 1st part of setup
36     vt_clrscr (); // clear the screen
37     set_statusline (UNAME);
38     printf ("%s (non-LP version) Build: %s\n", UNAME, BUILDDATE);
39     setup_irqs_and_faults (); // interrupt and fault handlers
40     keyboard_install (); // keyboard handler

```

```

41  initialize_timer ();           // timer handler
42  initialize_memory_late ();     // paging, 2nd part of setup
43  initialize_terminals ();      // ten virtual consoles
44  install_all_syscall_handlers (); // load syscall table
45  install_filesystem();         // fs: enable serial disk, fd, hd, swap
46  initialize_module ();         // external module (if available)
47  asm ("sti");                  // enable interrupts
48  #ifdef START_KERNEL_SHELL
49      kernel_shell ();
50  #endif
51  printf ("Starting five shells on tty0..tty4. "
52          "Press [Ctrl-L] for de/en keyboard.\n");
53  start_program_from_disk ("/init"); // load flat binary of init
54  // never reach this line!
55  }

```

Listing A.5: syscall.c

```

1  /*
2  Copyright (c) 2008-2014 Felix Freiling, University of Erlangen-Nürnberg, Germany
3  Copyright (c) 2011-2014 Hans-Georg Eßer, University of Erlangen-Nürnberg,
   Germany
4
5  This program is free software: you can redistribute it and/or modify it under
6  the terms of the GNU General Public License as published by the Free Software
7  Foundation, either version 3 of the License, or (at your option) any later
   version.
8
9  This program is distributed in the hope that it will be useful, but WITHOUT ANY
10 WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
11 PARTICULAR PURPOSE. See the GNU General Public License for more details.
12
13 You should have received a copy of the GNU General Public License along with
14 this program. If not, see <http://www.gnu.org/licenses/>. */
15
16
17 // this is syscall.c
18 #include "ulix.h"
19 #include "tables.c"
20 #include "irq.h"
21 #include "globals.h"
22
23
24 // install_syscall_handler() is used for installing a single
25 // system call handler function in the syscall table.
26
27 void
28 install_syscall_handler (int syscallno, void *syscall_handler) {
29     if (syscallno >= 0 && syscallno < MAX_SYSCALLS)
30         syscall_table[syscallno] = syscall_handler;

```

```

31 }
32
33
34 // syscall_handler() is the generic system call handler which
35 // is called when a process executes the "int 0x80" instruction.
36
37 void
38 syscall_handler (context_t *r) {
39     void (*handler) (context_t*); // handler is a function pointer
40     int number = r->eax;
41     if (number != __NR_get_errno) set_errno (0); // default: no error
42     handler = syscall_table[number];
43     if (handler != 0) handler (r);
44     else
45         printf ("Unknown syscall no. eax=0x%x; ebx=0x%x. eip=0x%x, esp=0x%x. "
46                "Continuing.\n", r->eax, r->ebx, r->eip, r->esp);
47 }
48
49
50
51 // handle errno
52
53 int
54 get_errno () {
55     if (scheduler_is_active) return thread_table[current_task].error;
56     else return startup_errno;
57 }
58
59 void
60 set_errno (int err) {
61     if (scheduler_is_active) thread_table[current_task].error = err;
62     else startup_errno = err;
63 }
64
65
66
67 // setup syscall handlers
68
69 void
70 install_all_syscall_handlers () {
71     install_syscall_handler (__NR_brk, syscall_sbrk);
72     install_syscall_handler (__NR_get_errno, syscall_get_errno);
73     install_syscall_handler (__NR_set_errno, syscall_set_errno);
74     install_syscall_handler (__NR_fork, syscall_fork);
75     install_syscall_handler (__NR_exit, syscall_exit);
76     install_syscall_handler (__NR_waitpid, syscall_waitpid);
77     install_syscall_handler (__NR_resign, syscall_resign);
78     install_syscall_handler (__NR_gettid, syscall_gettid);
79     install_syscall_handler (__NR_getpid, syscall_getpid);

```

```
80  install_syscall_handler (__NR_getppid, syscall_getppid);
81  install_syscall_handler (__NR_getpsinfo, syscall_getpsinfo);
82  install_syscall_handler (__NR_setpsname, syscall_setpsname);
83  install_syscall_handler (__NR_execve, syscall_execv);
84  install_syscall_handler (__NR_pthread_create, syscall_pthread_create);
85  install_syscall_handler (__NR_pthread_exit, syscall_pthread_exit);
86  install_syscall_handler (__NR_idle, syscall_idle);
87  install_syscall_handler (__NR_page_out, syscall_page_out);
88  install_syscall_handler (__NR_get_free_frames, syscall_get_free_frames);
89  install_syscall_handler (__NR_free_a_frame, syscall_free_a_frame);
90  install_syscall_handler (__NR_setterm, syscall_setterm);
91  install_syscall_handler (__NR_clrscr, syscall_clrscr);
92  install_syscall_handler (__NR_get_xy, syscall_get_xy);
93  install_syscall_handler (__NR_set_xy, syscall_set_xy);
94  install_syscall_handler (__NR_read_screen, syscall_read_screen);
95  install_syscall_handler (__NR_write_screen, syscall_write_screen);
96  install_syscall_handler (__NR_pthread_mutex_init, syscall_pthread_mutex_init);
97  install_syscall_handler (__NR_pthread_mutex_lock, syscall_pthread_mutex_lock);
98  install_syscall_handler (__NR_pthread_mutex_trylock,
    syscall_pthread_mutex_trylock);
99  install_syscall_handler (__NR_pthread_mutex_unlock,
    syscall_pthread_mutex_unlock);
100 install_syscall_handler (__NR_pthread_mutex_destroy,
    syscall_pthread_mutex_destroy);
101 install_syscall_handler (__NR_readchar, syscall_readchar);
102 install_syscall_handler (__NR_open, syscall_open);
103 install_syscall_handler (__NR_stat, syscall_stat);
104 install_syscall_handler (__NR_close, syscall_close);
105 install_syscall_handler (__NR_read, syscall_read);
106 install_syscall_handler (__NR_write, syscall_write);
107 install_syscall_handler (__NR_lseek, syscall_lseek);
108 install_syscall_handler (__NR_isatty, syscall_isatty);
109 install_syscall_handler (__NR_mkdir, syscall_mkdir);
110 install_syscall_handler (__NR_rmdir, syscall_rmdir);
111 install_syscall_handler (__NR_readdir, syscall_getdent);
112 install_syscall_handler (__NR_truncate, syscall_truncate);
113 install_syscall_handler (__NR_ftruncate, syscall_ftruncate);
114 install_syscall_handler (__NR_link, syscall_link);
115 install_syscall_handler (__NR_unlink, syscall_unlink);
116 install_syscall_handler (__NR_symlink, syscall_symlink);
117 install_syscall_handler (__NR_readlink, syscall_readlink);
118 install_syscall_handler (__NR_getcwd, syscall_getcwd);
119 install_syscall_handler (__NR_chdir, syscall_chdir);
120 install_syscall_handler (__NR_diskfree, syscall_diskfree);
121 install_syscall_handler (__NR_sync, syscall_sync);
122 install_syscall_handler (__NR_kill, syscall_kill);
123 install_syscall_handler (__NR_signal, syscall_signal);
124 install_syscall_handler (__NR_setuid32, syscall_setuid);
125 install_syscall_handler (__NR_setgid32, syscall_setgid);
```

```

126     install_syscall_handler (__NR_setreuid32, syscall_seteuid);
127     install_syscall_handler (__NR_setregid32, syscall_setegid);
128     install_syscall_handler (__NR_login, syscall_login);
129     install_syscall_handler (__NR_query_ids, syscall_query_ids);
130     install_syscall_handler (__NR_chown, syscall_chown);
131     install_syscall_handler (__NR_chmod, syscall_chmod);
132 }
133
134
135
136 // the system call implementations
137
138 void
139 syscall_sbrk (context_t *r) {
140     // ebx: increment
141     eax_return ( u_sbrk (r->ebx) );
142 }
143
144 void
145 syscall_get_errno (context_t *r) {
146     eax_return ( get_errno () );
147 }
148
149 void
150 syscall_set_errno (context_t *r) {
151     set_errno ((int)r->ebx);
152 }
153
154 void
155 syscall_fork (context_t *r) {
156     eax_return ((unsigned int) u_fork (r));
157 }
158
159 void
160 syscall_exit (context_t *r) {
161     // exit_code is in ebx register:
162     mutex_lock (thread_list_lock);
163     // close_open_files
164     thread_id pid = thread_table[current_task].pid;
165     int gfd;
166     for (int pfd = 0; pfd < MAX_PFD; pfd++) {
167         if ((gfd = thread_table[pfd].files[pfd]) != -1) u_close (gfd);
168     }
169
170     // modify thread table
171     thread_table[current_task].exitcode = r->ebx; // store exit_code
172     thread_table[current_task].state = TSTATE_EXIT; // mark process as finished
173     remove_from_ready_queue (current_task); // remove it from ready
         queue

```

```

174     wake_waiting_parent_process (current_task);           // wake parent
175     destroy_address_space (current_as);                   // return the memory
176
177     for (int pid = 0; pid < MAX_THREADS; pid++)
178         if (thread_table[pid].ppid == current_task)
179             thread_table[pid].ppid = 1; // set parent to idle_process
180                                           // notify children
181     mutex_unlock (thread_list_lock);
182
183     // finally: call scheduler to pick a different task
184     scheduler (r, SCHED_SRC_RESIGN);
185 }
186
187 void
188 syscall_waitpid (context_t *r) {
189     // ebx: pid of child to wait for
190     // ecx: pointer to status
191     // edx: options (ignored)
192     int chpid = r->ebx; // child we shall wait for
193
194     // check errors
195     if (chpid < 1 || chpid >= MAX_THREADS || thread_table[chpid].state == 0)
196         eax_return (-1); // error
197     if (!thread_table[chpid].used) eax_return (-1); // no such process
198     if (thread_table[chpid].ppid != current_task)
199         eax_return (-1); // not a child of mine
200
201     int *status = (int*)r->ecx; // address for the status
202     mutex_lock (thread_list_lock);
203     thread_table[current_task].waitfor = chpid;
204     block (&waitpid_queue, TSTATE_WAITFOR);
205     mutex_unlock (thread_list_lock);
206     syscall_resign (r); // here we resign_
207     *status = thread_table[chpid].exitcode;
208     thread_table[chpid].used = false; // finally remove child process
209     eax_return (chpid); // set the return value
210 }
211
212 void
213 syscall_resign (context_t *r) {
214     resign_calls++; // debug
215     inside_resign = true; // note: we have just started resigning and
216                           // don't want to be interrupted by the scheduler
217     scheduler (r, SCHED_SRC_RESIGN);
218     inside_resign = false;
219 }
220
221 void
222 syscall_gettid (context_t *r) {

```

```

223     eax_return (current_task);
224 }
225
226 void
227 syscall_getpid (context_t *r) {
228     eax_return (current_pid);
229 }
230
231 void
232 syscall_getppid (context_t *r) {
233     eax_return (current_ppid);
234 }
235
236 void
237 syscall_getpsinfo (context_t *r) {
238     unsigned int retval, pid;
239     // ebx: thread ID
240     // ecx: address of TCB block
241     pid = r->ebx;
242     if (pid > MAX_THREADS || pid < 1) {      // legal argument?
243         retval = 0; goto end;
244     }
245     if (thread_table[pid].used == false) {   // do we have this thread?
246         retval = 0; goto end;
247     }
248
249     // found a process: copy its TCB
250     memcpy ((char*)r->ecx, &thread_table[pid], sizeof (TCB));
251     retval = r->ecx;
252
253     end: eax_return (retval);
254 }
255
256 void
257 syscall_setpsname (context_t *r) {
258     strncpy (thread_table[current_task].cmdline,
259             (char*)r->ebx, CMDLINE_LENGTH-1);
260 }
261
262 void
263 syscall_execv (context_t *r) {
264     // generate command line in one string
265     char *path = (char*)r->ebx;           // path argument of execv()
266     char **argv = (char**)r->ecx;         // argv argument of execv()
267     int i = 0; char cmdline[CMDLINE_LENGTH] = "";
268     while (argv[i] != 0) {
269         strncpy (cmdline + strlen(cmdline), argv[i],
270                 CMDLINE_LENGTH-strlen(cmdline)-1);
271         strncpy (cmdline + strlen(cmdline), " ",

```

```

272         CMDLINE_LENGTH-strlen(cmdline)-1);
273     i++;
274 }
275 if (cmdline[strlen(cmdline)-1] == ' ')
276     cmdline[strlen(cmdline)-1] = '\\0';    // remove trailing blank
277
278 // call u_execv()
279 memaddress stack;
280 memaddress startaddr =
281     (memaddress) u_execv (path, argv, &stack); // sets stack
282 if (startaddr == -1)    eax_return (-1); // error
283
284 // update context and process commandline
285 r->eip    = startaddr;                // start_ running at address e_entry
286 r->useresp = (memaddress)stack;        // from ELF header
287 r->ebp    = (memaddress)stack;
288 strncpy (thread_table[current_task].cmdline, cmdline, CMDLINE_LENGTH);
289 }
290
291 void
292 syscall_pthread_create (context_t *r) {
293     // ebx: address of thread function
294     memaddress address = r->ebx;
295     u_pthread_create (NULL, NULL, address, NULL);
296 }
297
298 void
299 syscall_idle (context_t *r) {
300     asm ("sti"); // must be on, otherwise system will hang
301     asm ("hlt");
302 }
303
304 void
305 syscall_page_out (context_t *r) {
306     // ebx: page number
307     eax_return (page_out (current_as, r->ebx) );
308 }
309
310 void
311 syscall_get_free_frames (context_t *r) {
312     // no parameters
313     mutex_lock (swapper_lock);    // lock, will be freed in
314                                   // timer handler (timer.c)
315     eax_return (free_frames);
316 }
317
318 void
319 syscall_free_a_frame (context_t *r) {
320     // no parameters

```

```

321     addr_space_id pick_as      = -1;
322     int            pick_pageno, themin;
323     while (!mutex_try_lock (paging_lock)) ;    // active waiting for lock
324     for (int as = 1; as < MAX_ADDR_SPACES; as++) {
325         if (address_spaces[as].status == AS_USED) {
326             page_directory *pd = address_spaces[as].pd;
327             for (int i = 0; i < 768; i++) {      // < 768: only work on process
328                 memory
329                 if (pd->ptds[i].present) {      // directory entry in use
330                     page_table *pt = (page_table*) (PHYSICAL ((pd->ptds[i].frame_addr) <<
331                     12));
332                     for (int j = 0; j < 1024; j++) {
333                         if (pt->pds[j].present) {      // table entry in use
334                             int pageno = i*1024 + j;
335                             int index = hash ((as << 20) | pageno, PG_MAX_COUNTERS);
336                             if (pick_as == -1 && counter_table[index].used) {
337                                 // initialize minimum, pick
338                                 pick_as      = as;
339                                 pick_pageno = pageno;
340                                 themin = counter_table[index].count;
341                             } else {
342                                 if (counter_table[index].count < themin) {
343                                     themin = counter_table[index].count;
344                                     pick_as      = as;
345                                     pick_pageno = pageno;
346                                 }
347                             }
348                         }
349                     }
350                 }
351             }
352         }
353     }
354     mutex_unlock (paging_lock);
355
356     if (pick_as != -1) {
357         mutex_lock (paging_lock);
358         page_out (pick_as, pick_pageno);
359         mutex_unlock (paging_lock);
360     } else {
361         printf ("\nERROR: cannot pick a page to evict!\n");
362     }
363 }
364
365 void
366 syscall_clrscr (context_t *r) {
367     // no parameters, no return value
368     vt_clrscr ();
369 }

```

```
368
369 void
370 syscall_get_xy (context_t *r) {
371     // ebx: address of x position (char)
372     // ecx: address of y position (char)
373     vt_get_xy ((char*)r->ebx, (char*)r->ecx);
374 }
375
376 void
377 syscall_set_xy (context_t *r) {
378     // ebx: x position (char)
379     // ecx: y position (char)
380     vt_set_xy ((char)r->ebx, (char)r->ecx);
381 }
382
383 void
384 syscall_read_screen (context_t *r) {
385     // ebx: buffer address
386     read_write_screen ((char *) r->ebx, true);
387 }
388
389 void syscall_write_screen (context_t *r) {
390     // ebx: buffer address
391     read_write_screen ((char *) r->ebx, false);
392 }
393
394 void
395 syscall_pthread_mutex_init (context_t *r) {
396     // ebx: mutex id
397     // ecx: attributes, not implemented
398     eax_return ( u_pthread_mutex_init ( (pthread_mutex_t*)r->ebx,
399                                         (pthread_mutexattr_t*)r->ecx ) );
400 }
401
402 void
403 syscall_pthread_mutex_lock (context_t *r) {
404     // ebx: mutex id
405     eax_return ( u_pthread_mutex_lock ((pthread_mutex_t*)r->ebx) );
406 }
407
408 void
409 syscall_pthread_mutex_trylock (context_t *r) {
410     // ebx: mutex id
411     eax_return ( u_pthread_mutex_trylock ((pthread_mutex_t*)r->ebx) );
412 }
413
414 void
415 syscall_pthread_mutex_unlock (context_t *r) {
416     // ebx: mutex id
```

```

417     eax_return ( u_pthread_mutex_unlock ((pthread_mutex_t*)r->ebx) );
418 }
419
420 void
421 syscall_pthread_mutex_destroy (context_t *r) {
422     // ebx: mutex id
423     eax_return ( u_pthread_mutex_destroy ((pthread_mutex_t*)r->ebx) );
424 }
425
426 void
427 syscall_readchar (context_t *r) {
428     char c;
429     int t = thread_table[current_task].terminal;
430     terminal_t *vt = &terminals[t];
431
432     // get character, return 0 if there is no new character in the buffer
433     if (vt->kbd_count > 0) {
434         vt->kbd_count--;
435         vt->kbd_lastread = (vt->kbd_lastread+1) % SYSTEM_KBD_BUFLen;
436         c = vt->kbd[vt->kbd_lastread];
437     } else {
438         c = 0;
439         if ((current_task > 1) && scheduler_is_active) {
440             mutex_lock (thread_list_lock);
441             block (&keyboard_queue, TSTATE_WAITKEY);
442             mutex_unlock (thread_list_lock);
443
444             asm (".intel_syntax noprefix; \
445                 mov eax, 66;; \
446                 int 0x80;; \
447                 .att_syntax; ");
448         }
449     };
450     r->ebx = c;      // return value in ebx
451 }
452
453 void
454 syscall_open (context_t *r) {
455     eax_return ( gfd2pfd (u_open ((char*) r->ebx, r->ecx, 0) ) );
456 }
457
458 void
459 syscall_stat (context_t *r) {
460     eax_return ( u_stat ((char*) r->ebx, (struct stat*) r->ecx) );
461 }
462
463 void
464 syscall_getdent (context_t *r) {
465     // ebx: path, ecx: index, edx: dir_entry buffer

```

```

466     eax_return ( u_getdent ((char*) r->ebx, r->ecx,
467                             (struct dir_entry*) r->edx) );
468 }
469
470 void
471 syscall_close (context_t *r) {
472     // ebx: fd
473     int pfd = r->ebx;
474     thread_id pid = thread_table[current_task].pid;
475     r->eax = u_close (pfd2gfd (pfd));           // close_ (globally)
476     if (pfd >= 0 && pfd < MAX_PFD)
477         thread_table[pid].files[pfd] = -1;      // close_ (locally)
478 }
479
480 void
481 syscall_read (context_t *r) {
482     // ebx: fd, ecx: *buf, edx: nbytes
483     eax_return ( u_read (pfd2gfd (r->ebx), (byte*) r->ecx, r->edx) );
484 }
485
486 void
487 syscall_write (context_t *r) {
488     // ebx: fd, ecx: *buf, edx: nbytes
489     eax_return ( u_write (pfd2gfd (r->ebx), (byte*) r->ecx, r->edx) );
490 }
491
492 void
493 syscall_lseek (context_t *r) {
494     // ebx: fd, ecx: offset, edx: whence
495     eax_return ( u_lseek (pfd2gfd (r->ebx), r->ecx, r->edx) );
496 }
497
498 void
499 syscall_isatty (context_t *r) {
500     // ebx: file descriptor
501     eax_return ( pfd2gfd (u_isatty (r->ebx)) );
502 }
503
504 void
505 syscall_mkdir (context_t *r) {
506     // ebx: name of new directory, ecx: mode
507     eax_return ( u_mkdir ((char*)r->ebx, r->ecx) );
508 }
509
510 void
511 syscall_rmdir (context_t *r) {
512     // ebx: name of directory that we want to delete
513     eax_return ( u_rmdir ((char*)r->ebx) );
514 }

```

```
515
516 void
517 syscall_truncate (context_t *r) {
518     // ebx: filename, ecx: length
519     eax_return ( u_truncate ((char*)r->ebx, r->ecx) );
520 }
521
522 void
523 syscall_ftruncate (context_t *r) {
524     // ebx: file descriptor, ecx: length
525     eax_return ( u_ftruncate ( pfd2gfd (r->ebx), r->ecx) );
526 }
527
528 void
529 syscall_link (context_t *r) {
530     // ebx: original name, ecx: new name
531     eax_return ( u_link ((char*)r->ebx, (char*)r->ecx) );
532 }
533
534 void
535 syscall_unlink (context_t *r) {
536     // ebx: pathname
537     eax_return ( u_unlink ((char*)r->ebx) );
538 }
539
540 void
541 syscall_symlink (context_t *r) {
542     // ebx: target file name, ecx: symbolic link name
543     eax_return ( u_symlink ((char*)r->ebx, (char*)r->ecx) );
544 }
545
546 void
547 syscall_readlink (context_t *r) {
548     // ebx: file name
549     // ecx: buffer for link target
550     // edx: buffer length
551     eax_return ( u_readlink ((char*)r->ebx, (char*)r->ecx, r->edx) );
552 }
553
554 void
555 syscall_getcwd (context_t *r) {
556     // ebx: buffer for directory
557     // ecx: maximum length of path
558     eax_return ( u_getcwd ((char*)r->ebx, r->ecx) );
559 }
560
561 void
562 syscall_chdir (context_t *r) {
563     // ebx: new directory
```

```

564     eax_return ( u_chdir ((char*)r->ebx) );
565 }
566
567 void
568 syscall_diskfree (context_t *r) {
569     // ebx: address of diskfree query structure
570     mx_diskfree ((struct diskfree_query*)r->ebx);
571 }
572
573 void
574 syscall_sync (context_t *r) {
575     // this syscall takes no arguments
576     buffer_sync (1); // with lock_
577 }
578
579 void
580 syscall_kill (context_t *r) {
581     // ebx: pid of child to send a s signal, ecx: signal number
582     int retval; int target_pid = r->ebx; int signo = r->ecx;
583
584     if (!thread_table[target_pid].used) { // check if target process exists
585         // target process does not exist
586         set_errno (ESRCH);
587         retval = -1; goto end;
588     }
589
590     if (signo < 0 || signo > 31) { // check if signal is in range
591         // 0..31
592         set_errno (EINVAL);
593         retval = -1; goto end;
594     }
595
596     // check if current process may send a signal
597     if ((thread_table[current_task].euid == 0) ||
598         (thread_table[target_pid].euid == thread_table[current_task].euid)) {
599         retval = u_kill (target_pid, signo);
600     } else {
601         set_errno (EPERM);
602         retval = -1;
603     }
604     end: r->eax = retval;
605
606     // run scheduler if this was a raise operation
607     if (current_task == target_pid) {
608         asm (".intel_syntax noprefix; \
609             mov eax, 66;; \
610             int 0x80;; \
611             .att_syntax; ");

```

```
612 }
613
614 void
615 syscall_signal (context_t *r) {
616     // ebx: signal_ number
617     // ecx: address of signal_ handler
618     int signo = r->ebx;
619     sighandler_t func = (sighandler_t)r->ecx;
620     func = u_signal (signo, func);
621     eax_return (func);
622 }
623
624 void
625 syscall_setuid (context_t *r) {
626     // ebx: uid
627     eax_return ( u_setuid (r->ebx) );
628 }
629
630 void
631 syscall_setgid (context_t *r) {
632     // ebx: gid
633     eax_return ( u_setgid (r->ebx) );
634 }
635
636 void
637 syscall_seteuid (context_t *r) {
638     // ebx: euid
639     eax_return ( u_seteuid (r->ebx) );
640 }
641
642 void
643 syscall_setegid (context_t *r) {
644     // ebx: egid
645     eax_return ( u_setegid (r->ebx) );
646 }
647
648 void
649 syscall_login (context_t *r) {
650     // ebx: uid, ecx: password
651     eax_return ( u_login (r->ebx, (char*)r->ecx) );
652 }
653
654 void
655 syscall_query_ids (context_t *r) {
656     // ebx: type of ID
657     switch (r->ebx) {
658         case QUERY_UID:  eax_return (thread_table[current_task].uid);
659         case QUERY_EUID: eax_return (thread_table[current_task].euid);
660         case QUERY_GID:  eax_return (thread_table[current_task].gid);
```

```
661     case QUERY_EGID: eax_return (thread_table[current_task].egid);
662     default:         eax_return (-1);
663 }
664 }
665
666 void
667 syscall_chown (context_t *r) {
668     // ebx: path, ecx: owner, edx: group
669     eax_return ( u_chown ((char *)r->ebx, r->ecx, r->edx) );
670 }
671
672 void
673 syscall_chmod (context_t *r) {
674     // ebx: path, ecx: new mode
675     eax_return ( u_chmod ((char *)r->ebx, r->ecx) );
676 }
677
678 void
679 syscall_print_inode (context_t *r) {
680     int ino = r->ebx; // requested inode
681     printf ("syscall; ino = %d\n", ino);
682
683     struct minix2_inode in;
684     mx_read_inode (DEV_HDA, ino, &in);
685     printf ("i_mode: 0%o\n", in.i_mode);
686     printf ("i_nlinks: %d\n", in.i_nlinks);
687     printf ("i_size: %d\n", in.i_size);
688     printf ("i_zone:  [");
689     for (int i = 0; i < 7; i++) printf ("%d, ", in.i_zone[i]); printf ("]\n");
690 }
```