Handling Interrupts in the ULIX Operating System

(with classical code documentation)

Hans-Georg Eßer

May 27, 2015

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

This document contains a selected chapter from the introductory operating systems book "The Design and Implementation of the Ulix Operating System" [EF15]. In this excerpt we only treat the handling of interrupts and faults on an Intel-i386-based machine.

The Ulix source code consists of several C files and one Assembler file. These are compiled or assembled and the resulting object files are linked to create the kernel binary. The source directory contains the following files:

- `ulix.c`: kernel's `main()` function
- `globals.h`: global variables (with kernel-wide visibility)
- `block.c`: `readblock()`, `writeblock()` functions and device-specific implementations for floppy and hard disk controllers
- `fs.c`: logical filesystems (Minix, DevFS)
- `irq.c`: interrupt and fault handlers
- `keyboard.c`: keyboard driver
- `kshell.c`: kernel mode shell (debugging)
- `memory.c`: memory management (paging), paging in and out
- `module.c`: extensions (empty; used for Bachelor's theses)
- `queues.c`: management of ready and blocked queues
- `sched.c`: scheduler, context switch
- `serial.c`: serial ports (used for debugging)
- `string.c`: string functions (`strcpy()`, `strcmp()` etc.)
- `sync.c`: mutexes and semaphores

- `syscall.c`: treatment of system calls and all individual system call handlers
- `tables.c`: data (e. g. standard mount table, keyboard map)
- `terminal.c`: handling the ten text consoles; `printf()`
- `threads.c`: processes and threads, `fork()`, `exit()`, `pthread_create()` etc.
- `timer.c`: configuration of the timer chip, timer interrupt handler
- `start.asm`: assembler code

Complete listings of the files which are relevant for this excerpt (`irq.h`, `irq.c`, `start.asm`, `ulix.c`, `syscall.c`) can be found in Appendix A which begins on page 23.

Initialization of the interrupt and fault handlers occurs early in the system's overall initialization process: In line 39, the kernel's `main()` function

in `ulix.c`

```
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  }
```

calls `setup_irq_and_faults()` (in `irq.c`) which is what this book excerpt focuses on.

2

Interrupts and Faults

All modern CPUs and even many of the older ones such as the Zilog Z80 8-bit processor can be interrupted: the CPU has an input line which can be triggered by an external device connected to this line. When such an interrupt occurs, the current activity is suspended, and the CPU continues operation at a specified address: it executes an interrupt handler.

In principle a device could be directly connected to the CPU, but modern machines contain many devices which want to interrupt the processor, e. g. the disk controllers, the keyboard controller, the serial ports, or the on-board clock. Thus an extra device, called the *interrupt controller*, intermediates between the other devices and the CPU. One of the advantages of such an interrupt controller is that it is programmable: it is possible to enable or disable specific interrupts whereas the CPU itself can only completely enable or disable all interrupts, using the `sti` (set interrupt flag) and `cli` (clear interrupt flag) instructions. (These machine instructions exist on Intel-x86-compatible CPUs; other chips have similar instructions.) Being programmable also means that interrupt numbers can be remapped (we will see later why this is helpful). Interrupt controllers with these features are called *programmable interrupt controllers* (PICs), and we'll use that abbreviation throughout the rest of this chapter.

interrupt controller

`sti`, `cli`

PICs

After the implementation of interrupts we will also take a look at fault handling since the involved mechanisms are very similar to those which we need for handling interrupts. The main difference between interrupts and faults is that faults occur as a direct consequence of some specific instruction that our code executes. In that sense they are *synchronous*. Interrupts on the other hand occur without any connection to the currently executing instruction, since they are not triggered (immediately) by our code but by some device. That is why they are called *asynchronous*.

synchronous

asynchronous

2.1 Examples for Interrupt Usage

Interrupt handling is a core functionality which is used in lots of places: without interrupts we would not be able to build a useful operating system.

Let's look at some example features of Ulix which depend heavily on interrupts:

Multi-tasking Ulix can execute several processes in parallel and switch between them using a simple round-robin scheduling mechanism. That is only possible because the clock chip on the motherboard regularly generates timer interrupts, and Ulix installs a timer interrupt handler which—when activated—calls the scheduler to check whether it is time to switch to a different process. If there were no interrupts, we could only implement *non-preemptive scheduling* which relies on the processes to give up the CPU voluntarily.

timer handler

Keyboard input Whenever you press or release a key on a PC, either event generates an interrupt. Ulix picks up these interrupts and the keyboard interrupt handler reads a key press or key release code from the controller.

keyboard handler

polling

A keyboard driver does not need interrupts, but the alternative is to constantly poll (query) the keyboard controller in order to find out whether a new event has occurred. That's possible but wastes a lot of CPU time. Polling does not work well in a multi-tasking environment. (However for a single-tasking operating system it may be good enough.)

Media Reading and writing hard disks and floppy disks also depends on interrupts: In the Ulix implementation of filesystems (and disk access) a process which wants to read or write makes a system call which sends a request to the drive controller. Then Ulix puts the calling process to sleep. Once the request has been served, the drive controller generates an interrupt, and the interrupt handler for the hard disk controller or the floppy disk controller (these are two separate handlers) deals with the data and wakes up the sleeping process.

ATA/FDC handlers

Again, this could be done without interrupts. But the process would have to remain active and continuously poll the controller to find out whether the data transfer has been completed.

Serial ports Finally, the serial ports are similar to the keyboard, since all of them are *character devices*: they transfer single bytes (instead of blocks of bytes).

2.2 Interrupt Handling on the Intel Architecture

Intel 8259 PIC

The classical IBM PC used the Intel 8259 Programmable Interrupt Controller, compatible descendents of which are still used in modern computers. The 8259 has eight input lines (through which up to eight separate devices may connect) and one output line which forwards received interrupt signals to the CPU. It is possible to use more than one 8259 PIC

since these controllers can be cascaded which means that a second controller's output pin is connected with one of the first controller's input pins (typically the one for device 2, see Figure 2.1). With that cascade, devices connected to the first controller keep their normal numbers (0, 1, 3–7 with 2 reserved for the second controller), and devices connected to the second controller use device numbers between 8 and 15, allowing for a total of 15 (= 16 − 1) separate device numbers. The first or primary controller is called *Master PIC*, the second one is the *Slave PIC* (as it is not directly connected to the CPU but relies on the master PIC to have its interrupts signals forwarded). The numbers 0–15 are called *Interrupt Request Numbers* (IRQs).

Cascading PICs

Master PIC

Slave PIC

IRQ

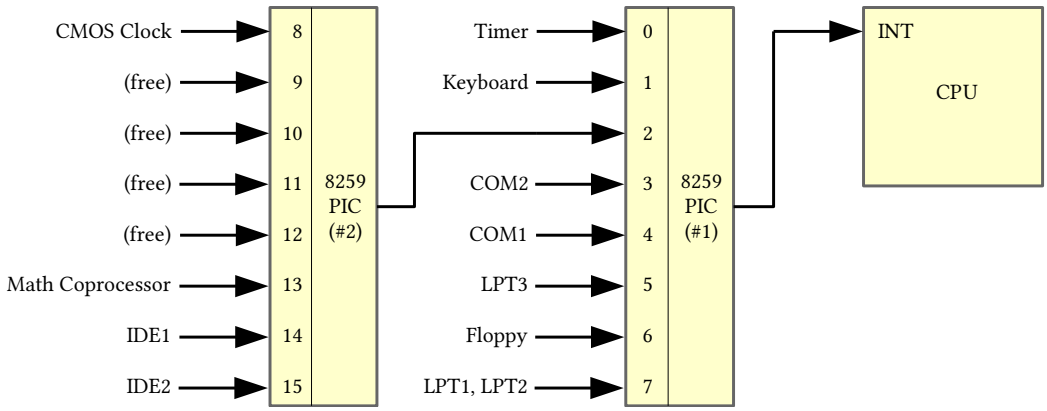

Figure 2.1: Two PICs are cascaded, which allows for 15 distinct interrupts.

As you can see from the figure, there is a fixed mapping of some devices to specific IRQs. We will use the following IRQs in the Ulix implementation:

- **0: Timer Chip**. On a PC's mainboard you can find a (programmable) timer chip which regularly generates interrupts. We will use timer interrupts to call the scheduler (besides other tasks).
- **1: Keyboard**. This is the interrupt generated by PS/2 keyboards. A USB keyboard would be handled differently, but we do not support USB devices.
- **2: Slave PIC**. As already mentioned, IRQ 2 is reserved for connecting the secondary (slave) PIC.
- **3: Serial Port 2**. The second serial port will be used for our implementation of what we've called the *serial hard disk*—it is discussed in the full Ulix book.
- **4: Serial Port 1**. We only use the first serial port for output (when running Ulix in a PC emulator), thus we will not install an interrupt handler for this IRQ.
- **6: Floppy**. This is the IRQ for the floppy controller. It can handle up to two floppy drives.
- **14: Primary IDE Controller**. And finally, 14 is the IRQ of the primary IDE controller. Many PC mainboards contain two controllers, with each of them allowing two drives to connect. The secondary IDE controller would generate the interrupt number 15,

but we're going to support only one controller.

Names for the IRQ numbers 0, 1, 2, 3, 4, 6 and 14 (`IRQ_TIMER`, `IRQ_KBD`, `IRQ_SLAVE`, `IRQ_COM2`, `IRQ_COM1`, `IRQ_FDC` and `IRQ_IDE`) are defined in lines 23–29 of `irq.h`.

2.2.1 Using Ports for I/O Requests

Going where? We want to initialize the PICs, which means directly talking to these controllers. Like with most other devices we can use the machine instructions `in` and `out` to find out the PIC's current status and tell it what to do.

Here we provide the code which lets us access the controllers.

I/O Ports Access to many hardware components (including the PICs) is possible via *I/O ports*. Using `in` and `out` machine instructions it is possible to transfer bytes, words or doublewords between a CPU register and a memory location or register on some device (such as a hard disk controller).

The Intel 80386 Programmer's Reference Manual [Int86, pp. 146–147] explains:

`in`, `out`

> "The I/O instructions `IN` and `OUT` are provided to move data between I/O ports and the *EAX* (32-bit I/O), the *AX* (16-bit I/O) or *AL* (8-bit I/O) general registers. `IN` and `OUT` instructions address I/O ports either directly, with the address of one of up to 256 port addresses coded in the instruction, or indirectly via the *DX* register to one of up to 64K port addresses.
>
> `IN` (Input from Port) transfers a byte, word or doubleword from an input port to *AL*, *AX* or *EAX*. If a program specifies *AL* with the `IN` instruction, the processor transfers 8 bits from the selected port to *AL*. If a program specifies *AX* with the `IN` instruction, the processor transfers 16 bits from the port to *AX*. If a program specifies *EAX* with the `IN` instruction, the processor transfers 32 bits from the port to *EAX*.
>
> `OUT` (Output to Port) transfers a byte, word or doubleword to an output port from *AL*, *AX* or *EAX*. The program can specify the number of the port using the same methods as the `IN` instruction."

For accessing 8-bit, 16-bit and 32-bit ports, the Intel assembler language provides separate commands `inb` / `outb` (byte), `inw` / `outw` (word) and `inl` / `outl` (long: doubleword) which make it explicit what kind of transfer is wanted. We'll use them in the functions `inportb`, `inportw`, `outportb` and `outportw`. There are several possible C implementations with inline assembler code, here are two examples from `irq.c`:

in `irq.c`

```
34 byte
35 inportb (word port) {
36   byte retval;
37   asm volatile ("inb %%dx, %%al" : "=a"(retval) : "d"(port));
38   return retval;
39 }
```

in irq.c

```
48 void
49 outportb (word port, byte data) {
50   asm volatile ("outb %%al, %%dx" : : "d" (port), "a" (data));
51 }
```

and the corresponding word-based functions `inportw` and `outportw` have similar implementations that use the `inw` and `outw` assembler instructions (see lines 34–56 of `irq.c`).

We could provide `inportl` and `outportl` (for 32-bit values) in a similar fashion, using `inl`, `outl` and the 32-bit register *EAX* (instead of the 16-bit and 8-bit versions *AX* and *AL*), but we do not need them. (Remember that *EAX*, *AX* and *AL* are (parts of) the same register, see Figure 2.2. On a 64-bit machine, *RAX* is the 64-bit extended version of *EAX*.)

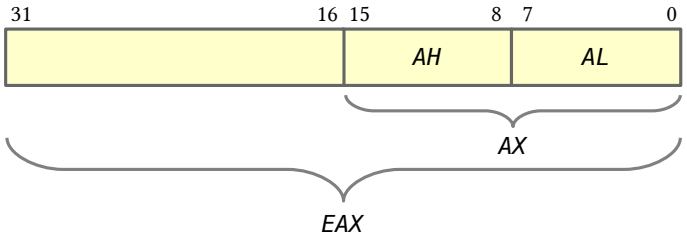

Figure 2.2: The lower half of *EAX* is *AX* which in turn is split into *AH* (high) and *AL* (low).

2.2.2 Initializing the PIC

Going where?

Now that we have functions for talking to devices we can set up the two PICs. We will configure one as master and the other as slave, and we also remap the interrupt numbers from 0–15 to 32–47 because the first 32 numbers are reserved for faults (see Section 2.3).

The PICs can be accessed via four ports with the port numbers `0x20`, `0x21`, `0xA0` and `0xA1` for which we declare the macro constants `IO_PIC_MASTER_CMD`, `IO_PIC_MASTER_DATA`, `IO_PIC_SLAVE_CMD` and `IO_PIC_SLAVE_DATA`, respectively (in lines 37–41 of `irq.h`).

They need to be initialized by sending them four "Initialization Command Words" (ICW) called ICW1, ICW2, ICW3 and ICW4 in a specific order, using specific ports. Each of the PICs has a command register and a data register. During normal operation we can write to the data register (using the ports `IO_PIC_MASTER_DATA` and `IO_PIC_SLAVE_DATA` for PIC1 or PIC2, respectively) to set the *interrupt mask*: That's a byte where each bit tells the controller whether it shall respond to a specific interrupt (1 means: mask, i. e., ignore the interrupt; 0 means: forward it to the CPU). We will start with an interrupt mask of `0xFF` for each controller (all bits are 1), thus all hardware interrupts will be ignored.

interrupt mask

The following code was taken from Bran's Kernel Development Tutorial [Fri05] (e. g. from the source file `irq.c` of that tutorial) and modified.

For programming the controller, we can send configuration data to the data port, but we have to initialize the programming by writing to the command port. The complete sequence is as follows:

- First we send ICW1 to both PICs. ICW1 is a byte whose bits have the following meaning [Int88, p. 11]:

  0 D_0: ICW4 needed? We set this to 1 since we want to program the controller.

  1 D_1: Single (1) / Cascade (0) mode: We set this to 0 since there's a slave.

  2 D_2: Call Address Interval (ignored), the default value is 0.

  3 D_3: Level (1) / Edge (0) Triggered Mode: we set this to 0.

  4 D_4: Initialization Bit: We set it to 1 because we want to initialize the controller.

  5,6,7 D_5, D_6, D_7: not used on x86 hardware, set to 0.

  This results in the byte `00010001` (`0x11`). The value is the same for both PICs. As mentioned before, ICW1 must be sent to the PICs' command registers.

remap the interrupts

- In the next step we send ICW2 to the PICs' data registers. The lowest three bits specify the offset for remapping the interrupts. Since the first 32 interrupts must be reserved for processor exception handlers (e. g. "Division by Zero" and "Page Fault" handlers), we map the interrupts 0–15 to the range 32–47 (`0x20` – `0x2f`).

  Each PIC would normally generate interrupts in the range 0–7, thus the offset is not the same for both PICs: For PIC1 it is `0x20` (32; mapping 0–7 to 32-39), and for PIC2 it is `0x28` (40; mapping 0–7 to 40–47).

- The next command word is ICW3. Its functionality depends on whether we send it to the master (PIC1) or the slave (PIC2): The PICs already know that they are master and slave (because we sent that information as part of ICW1) [Int88, p. 12].

  The master expects a command word byte in which each set bit specifies a slave connected to it. We have only one slave and want to make it signal new interrupts on interrupt line 2 of the master. Thus, only the third bit (from the right) must be set: $\texttt{00000100}_b$ = `0x04`.

  The slave needs a slave ID. We give it the ID 2 = `0x02`.

- To end the sequence, we send ICW4 which is just `0x01` for x86 processors [Int88, p. 12].

Finally, we set the initial interrupt mask to 0 (disabling all interrupts). It takes two write operations to the two PICs' data registers since each one handles only eight of the total 16 possible interrupts.

Thus, the whole initialization sequence happens as follows in `setup_irqs_and_faults()`:

in irq.c

```
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
```

IRQ mask

The last two lines of that code block set the *IRQ mask* to `0xFFFF` = 1111111111111111_b via `set_irqmask()` which disables (*masks*) all interrupts, and then enable (*unmask*) the interrupt for the slave PIC with `enable_interrupt()` —both functions have not been mentioned so far. The IRQ mask is a 16-bit value in which each bit says whether some interrupt is enabled (value 0) or not (value 1). We must talk to both PICs to set the mask, the master PIC gets the lower eight bits (for the interrupts 0–7), the slave PIC gets the upper eight bits (for the interrupts 8–15):

in irq.c

```
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  }
```

We can also read the mask from the two PICs with a similar function we call `get_irqmask()` in which we read the two data registers instead of writing them (lines 104–108 of `irq.c`).

In other chapters (of the Ulix book) we will often enable a specific interrupt for some device after we've prepared its usage, e. g. for the floppy controller. For that purpose, we will always use `enable_interrupt()` like we did above. It simply reads the current IRQ mask, clears a bit, and writes the new value back:

in irq.c

```
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 }
```

With the remapping in place we can now create entries for the interrupt handler table—we need some new data structures for them.

2.2.3 Interrupt Descriptor Table

Going where?

The PICs are initialized and will do the right thing when an interrupt occurs, but we haven't told the CPU yet what to do when it receives one. This calls for a new data structure, the *Interrupt Descriptor Table*, which we must define according to the Intel standards and fill with proper values.

While the first Intel-8086/8088-based personal computers used a fixed address in RAM to store the interrupt handler addresses, modern machines let us place the table anywhere in memory. After preparing the table we must use the machine instruction `lidt` (load interrupt descriptor table register) to tell the CPU where to search.

`lidt`

The procedure we need to follow is similar to the one for activating segmentation via a GDT:[1]

1. We first store interrupt descriptors (each of which is eight bytes large) in a table consisting of **`struct`** `idt_entry` entries,
2. then we create some kind of pointer structure **`struct`** `idt_ptr` which contains the length and the start address of the table,
3. and finally we execute `lidt` (compare this to `lgdt` which loads the GDT).

Figure 2.3 shows the layout of an IDT entry. The *Flags* halfbyte (second line, left in the figure) consists of

- the present flag (bit 3) which must always be set to 1,
- two bits (2 and 1) for the Descriptor Privilege Level (DPL). We will always set this to $11_b = 3$ since we want all interrupts to be available all the time (when we're in kernel or user mode) and
- a so-called "storage segment" flag (bit 0; which must be set to 0 for an "interrupt gate", see next entry).

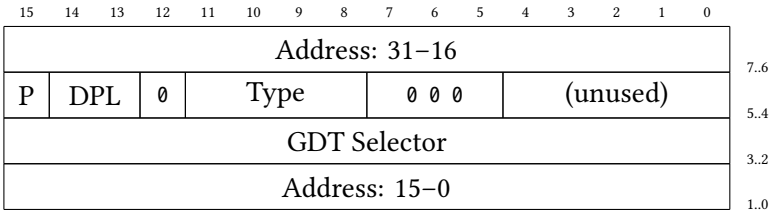

Figure 2.3: An Interrupt Descriptor contains the address of an interrupt handler and some configuration information.

The *Type* halfbyte declares what kind of descriptor this is: we will always set it to 1110_b, making this descriptor an

[1] The *global descriptor table* (GDT) is treated in detail in the full ULIX book; it contains segment descriptors which are required for enabling paging; overall we need to define four segments for data and code in user and kernel mode.

- *80386 32-bit interrupt gate* descriptor (which is what we want).

Besides this type, there are alternatives:

- 0101_b for an *80386 32-bit task gate*,
- 0110_b for an *80286 16-bit interrupt gate*,
- 0111_b for an *80286 16-bit trap gate* and
- 1111_b for an *80386 32-bit trap gate*,

but we will not go into the details about these. Instead of interrupt gates we could also use trap gates, the difference between those being that "for interrupt gates, interrupts are automatically disabled upon entry and reenabled upon `IRET` which restores the saved *`EFLAGS`*" [OSD13].

We will use the following datatype definitions:

in `irq.h`

```
58 struct idt_entry {
59     unsigned int addr_low  : 16;   // lower 16 bits of address
60     unsigned int gdtsel    : 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));
```

The *selector* must be the number of a code segment descriptor (in the GDT); we will always set this to `0x08` since our kernel (ring 0) code segment uses that number (see the code for installing a "flat GDT" which is only available in the full Ulix book).

The IDT pointer has the same structure as the GDT pointer (discussed in the full Ulix book): it informs about the length and the location of the IDT via its `limit` (16-bit) and `base` (32-bit) members (lines 71–74, `irq.h`).

In theory, an interrupt number can be any byte, i. e., a value between 0 and 255. We will use a full IDT with 256 entries even though most of the entries will be null descriptors—if somehow an interrupt is generated which has a null descriptor, the CPU will generate an "unhandled interrupt" exception. We will talk about exceptions right after we've finished the interrupt handling code.

The variables `idt` (an array of 256 `struct idt_entry` fields) and `idtp` (of type `struct idt_ptr`; both defined on lines 82–83 of `globals.h`) will now be used in a way that is similar to how we used `gdt` (a `struct gdt_entry` array) and `gp` (a `struct gdt_ptr` structure) when we wrote the GDT code (again, this information is only available in the Ulix book).

The function `fill_idt_entry()` writes an entry of the IDT:

in `irq.c`

```
72  void
73  fill_idt_entry (byte num, unsigned long address,
74      word gdtsel, 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].gdtsel    = gdtsel;           // GDT sel.: user or kernel mode?
79      idt[num].zeroes    = 0;
80      idt[num].flags     = flags;
81      idt[num].type      = type;
82    }
83  }
```

Parts of all of our interrupt handlers will be assembler code (which we store in `start.asm`); we'll explain soon why that has to be. In that file you can find 16 assembler functions named `irq0`, `irq1`, ..., `irq15` which are exported (declared as global in `start.asm`) and made available in the C code file by declaring them as external functions (`extern void irq0()`, `irq1()`, ...). We will store the function addresses in an array which simplifies accessing them:

in `globals.h`

```
84  void (*irqs[16])() = {
85    irq0, irq1, irq2,  irq3,  irq4,  irq5,  irq6,  irq7,      // store them in
86    irq8, irq9, irq10, irq11, irq12, irq13, irq14, irq15      // an array
87  };
```

Back in the `setup_irqs_and_faults()` function (that we already discussed when we explained programming the PICs), the following lines fill 16 entries (with indices 32–47) of the IDT:

```
1  for (int i = 0;  i < 16;  i++) {
2    fill_idt_entry (32 + i,
3                    (unsigned int)irqs[i],
4                    0x08,
5                    0b1110,     // flags: 1 (present), 11 (DPL 3), 0
6                    0b1110);    // type:  1110 (32 bit interrupt gate)
7  }
```

2.2.4 Writing the Interrupt Handler

Going where? Everything is prepared for interrupt handlers — now we need to define them, i. e., implement the `irq0()`, ... `irq15()` functions. This step requires some assembler code and some C code.

We have installed handlers for all 16 interrupts, but what do they do? We will define part of their code in the assembler file, but we start with a description of what we expect to happen in general.

When an interrupt occurs, the CPU suspends the currently running code, saves some information on the stack, and then jumps to the address that it finds in the IDT. (It also uses a different stack and switches to kernel mode if it was in user mode when the interrupt occurred.) Then the interrupt handler runs, and once it has finished its job, it returns with the `iret` instruction. `iret` is different from the regular `ret` instruction which normal functions use for returning to the calling function: it is the special "return from interrupt" instruction which restores the original state (user or kernel mode, stack, *EFLAGS* register) so that the regular code can continue as if the interrupt had never happened.

`iret`

Switching to the interrupt handler can mean a change of the privilege level that the CPU executes in: So far we've only let Ulix work in ring 0 (kernel mode), but later when we introduce processes (in the Ulix book) it can happen that an interrupt occurs while the CPU runs in ring 3 (user mode). If that is the case, the privilege level changes (from 3 to 0). When such a transition occurs, the information (return address etc.) is not written to the process' user mode stack, but on the process' kernel stack which is located elsewhere and normally used during the execution of *system calls*. For now, the relevant piece of information is that different information gets stored on the "target stack": In case of a privilege change the CPU first writes the contents of the *SS* and *ESP* registers on the (new) stack—this does not happen if the CPU was already operating in ring 0. Next, *EFLAGS*, *CS* and *EIP* are written to the stack: that is all we need for returning to the interrupted code. Figure 2.4 shows the different stack contents when the interrupt handler starts executing [Int86, p. 159].

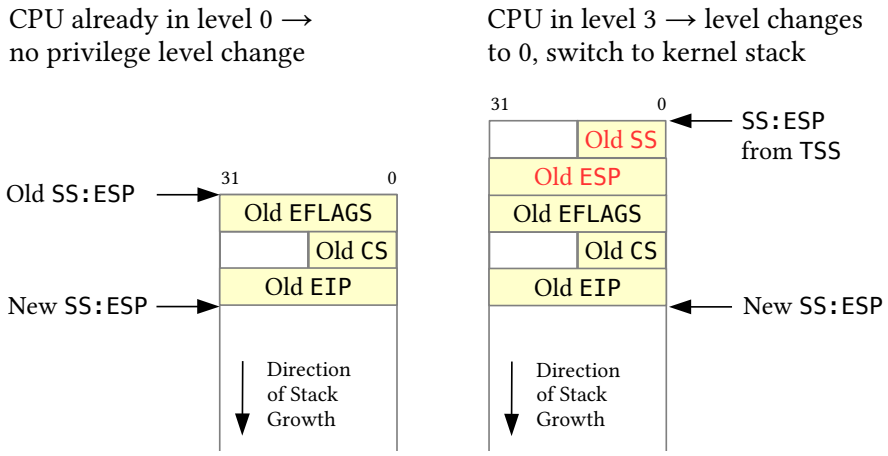

Figure 2.4: When entering the interrupt handler, the stack contains information for returning from the handler. Left: without privilege level change; right: with change from level 3 to 0, extra data marked red.

We cannot directly use a C function as an interrupt handler because once it would finish its work, it would do a regular `RET` which does not do what we want. (Of course we could use inline assembler code inside the C function to make it work anyway, but it makes more sense to directly implement parts of the handlers in assembler.)

2.2.4.1 The Context Data Structure

We want to be able to define handler functions in C which get called from the assembler code. Those functions will all have the following prototype:

```
void handler_function (context_t *r);
```

context

where `context_t` is a central data structure that can hold all the registers we use on the Intel machine. It will also be used in fault handlers, system call handlers and several other functions which need information about the current state.

We define the `context_t` structure so that it matches the way in which we set up the stack in the assembler part of the handler:

in ulix.h

```
624 typedef struct {
625   unsigned int gs, fs, es, ds;
626   unsigned int edi, esi, ebp, esp, ebx, edx, ecx, eax;
627   unsigned int int_no, err_code;
628   unsigned int eip, cs, eflags, useresp, ss;
629 } context_t;
```

2.2.4.2 Assembler Part of the Handler

In order to have a handler function see useful values in the structure that `r` points to, we need to push the register contents in the reverse order onto the stack:

```
pusha
push  ds
push  es
push  fs
push  gs
push  esp  ; pointer to the context_t
```

The first instruction `pusha` (push all general registers) pushes a lot of registers onto the stack: *EAX*, *ECX*, *EDX*, *EBX*, the old value of *ESP* (before the `pusha` execution began), *EBP*, *ESI*, and *EDI*—in that order. We add the segment registers *DS*, *ES*, *FS* and *GS*, and you can see that we've successfully handled the first two lines of the `context_t` type definition. When the interrupt occurred, the registers *EFLAGS*, *CS* and *EIP* (and possibly also *SS* and the user mode's *ESP*) were also pushed on the stack which gives us the values in the fourth line of the `context_t` definition.

What's missing are the values on the third line: We want to tell the handler *which interrupt* occurred so that we can use the same interrupt handler for several interrupts—for example, if we supported both IDE controllers (with interrupts 14 and 15) we could use that trick to run the same IDE handler when either of those interrupts occurred; thus, between the automatically happening push operations and the ones we perform in the code above we also push the interrupt number and another value `err_code` which can hold an error code. Interrupts don't have an error code, but we will recycle the same code later when we deal with faults, and some of those do provide an error code.

The final `push esp` statement in the code above is necessary because we cannot just place the structure contents on the stack: the handler function expects a pointer (`context_t *r`),

and ESP contains just that pointer: the start address of the structure. Figure 2.5 shows the layout of the stack after the assembler part has finished the preparations.

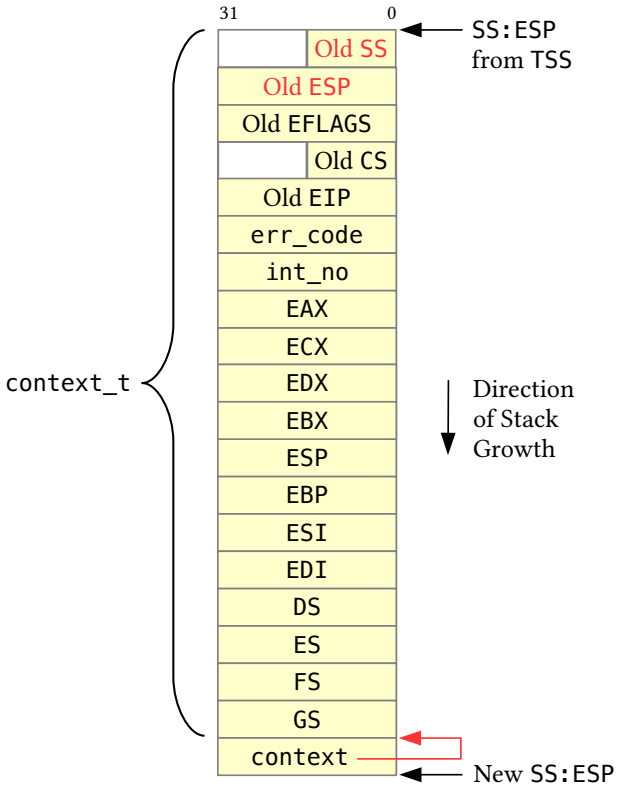

Figure 2.5: Stack after interrupt handler initialization by the assembler part.

Later, when the handler's task is completed, we will need to pop the registers from the stack—in the reverse order:

```
pop  esp
pop  gs
pop  fs
pop  es
pop  ds
popa
```

Now here's an example of how we could implement the interrupt handler for IRQ 15:

```
1  push byte 0           ; error code
2  push byte 15          ; interrupt number
3  pusha
4  push ds
5  push es
6  push fs
7  push gs
8  push esp              ; pointer to the context_t
9  call irq_handler      ; call C function
10 pop  esp
11 pop  gs
12 pop  fs
13 pop  es
14 pop  ds
15 popa
16 add  esp, 8           ; for errcode, irq no.
17 iret
```

This contains all we need:

1. The two `push` commands in lines 1–2 add the error code and the interrupt number (which is 15 in this example).
2. The further `push` commands in lines 3–8 complete the `context_t` data structure and also push a pointer to it.
3. Now the stack is prepared properly to call the C function `irq_handler` (line 9).
4. After returning, we first have to undo the `push` operations of lines 3–8 with the corresponding `pop` operations in lines 10–15.
5. Then we modify the stack address: we add 8 (line 16), thus undoing the two `push` operations for the error code and the interrupt number.
6. Finally we return from the handler with `iret` (line 17).

We need almost the same code 16 times (for IRQs 0 to 15)—the only difference between the 16 versions is the interrupt number that we push in the second instruction. We simplify our code by having our individual handlers just push the two values (0 and the interrupt number) and then jump to an address which provides the common commands. The 0 value is a placeholder for an error code which cannot occur in interrupt handlers, but (as mentioned before) we will also implement fault handlers which shall use the same stack layout, and some of them will write a fault-specific error code into that location.

The listing on the following page (containing lines 105–147 of `start.asm`) shows how the 16 assembler functions are implemented using a macro.

in start.asm

```
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
          :
```

in start.asm

```
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
```

Our interrupt handling code is a slightly improved version of the code which Bran's Kernel Tutorial [Fri05] uses; the original code contains some extra instructions that we don't need for the ULIX kernel.

2.2.4.3 C Part of the Handler

Finally, we show what happens when the assembler code calls the external handler function `irq_handler()` that we implement in the C file `irq.c`.

The first thing our handler needs to do is acknowledge the interrupt. For that purpose it sends the command

in irq.h

```
47 #define END_OF_INTERRUPT    0x20
```

to all PICs which are involved: In case of an interrupt number between 0 and 7 that is only the primary PIC; in case the number is 8 or higher, both controllers need to be informed.

Omitting this step would stop the controller from raising further interrupts which would basically disable interrupts completely.

Next we check whether a specific handler for the current interrupt has been installed in the `interrupt_handlers[]` array of interrupt handlers.

in irq.c

```
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
          (always)
215
216   handler = interrupt_handlers[number];
217   if (handler != NULL)  handler (r);
218 }
```

As a last step we provide a function `install_interrupt_handler()` which lets us enter (pointers to) handler functions in this array; it is pretty simple:

in irq.c

```
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 }
```

IDTR

Early during system initialization in the kernel's `main()` function we need to load the Interrupt Descriptor Table Register (*IDTR*) so that the CPU can find the table. The function `setup_irqs_and_faults()` does that (and more):

in irq.c

```
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 ();
        ⋮
```

lidt

It uses the assembler function `idt_load()` which writes the address of `idtp` to the *IDTR* register via the `lidt` instruction:

in start.asm

```
149 extern idtp                 ; defined in the C file
150 global idt_load
151 idt_load:
152         lidt [idtp]
153         ret
```

In other chapters of the Ulix book we will often use this function in commands similar to `install_interrupt_handler (IRQ_SOMEDEV, somedev_handler);`.

2.3 Faults

As we've mentioned in the introduction to this chapter, handling a fault is very similar to handling an interrupt. Since you've just seen the interrupt code, you will recognize many concepts at once while we present the fault handling code.

Like we defined the interrupt handlers `irq0()` to `irq15()` in the assembler file `start.asm`, we do the same with 32 fault handler functions `fault0()` to `fault31()` which are declared as external functions in `ulix.h`. We enter these in the IDT just like we did with the `irq*()` functions. Since there are so many of them, we'll use a macro that calls `fill_idt_entry()` with the same *flags* and *type* arguments as before:

in irq.c
```
129 #define FILL_IDT(i) \
130   fill_idt_entry (i, (unsigned int)fault##i, 0x08, 0b1110, 0b1110)
```

These macro calls occur in `setup_irqs_and_faults()` which we have already discussed earlier when we set up the interrupt handlers (see page 18).

in irq.c
```
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);
```

After macro expansion this will generate commands such as

```
fill_idt_entry (31,(unsigned)fault31, 0x08, 0b1110, 0b1110);
```

In the assembler file we use the same trick for the `fault*()` functions that you've just seen for `irq*()`:

in start.asm
```
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
```

The handlers all look similar: We push one or two bytes on the stack and then jump to `fault_common_stub`. The choice of one or two arguments depends on the kind of interrupt that occurred: for some faults the CPU pushes a one-byte error code on the stack, and for some others it does not. In order to have the same stack setup (regardless of the fault) we push an extra null byte in those cases where no error code is pushed.

error code

The code always looks like one of the following two cases:

```
fault5:  push byte 0  ; error code
         push byte 5
         jmp fault_common_stub
```

```
fault8:  ; no error code
         push byte 8
         jmp fault_common_stub
```

Since we do not want to type this repeatedly, we use `nasm`'s macro feature which lets us write simple macros for both cases. `fault_macro_0` handles the cases where we need to push

an extra null byte (as in `fault5` above), and `fault_macro_no0` handles the other cases (as in `fault8` above):

in start.asm

```
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
```

With these macros the rest is straight-forward:

in start.asm

```
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)
```

⋮

(and some more similar lines for the faults 20—31 which are also reserved).

`fault_common_stub` is—almost—a rewrite of `irq_common_stub`, the only difference is that we call a different C function `fault_handler()` in the middle which we need to declare as external in the assembler file:

in start.asm

```
205 extern fault_handler
```

The stub saves the processor state, calls the handler function and restores the stack frame:

in start.asm

```
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
```

Initially our fault handlers will just output a message stating the cause of the fault and then halt the system; later we will provide fault handlers for some types of faults which try to solve the problem and let the operation go on. The error messages are stored in an array `exception_messages[]` of strings (in `tables.c`, lines 22–36). We get the correct message by accessing the proper entry of the array, e. g., for a page fault (with fault number 14; `"Page Fault"`) it is stored in `exception_messages[14]`.

Fault Handler

Our C fault handler `void fault_handler (context_t *r)` displays some information about the problem and checks whether the fault occurred while a process was running (by testing whether `r->eip < 0xc0000000`). If not, the system switches to the kernel mode shell (and is broken).

The page fault handler is a special case: we expect to deal with page faults silently (via the `page_fault_handler()` function; see the page fault chapter of the Ulix book), so we check for this case before doing anything else.

in irq.c

```
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   }
```

For displaying the status information (lines 266–278) we look at the register contents which are provided by `r`. Especially interesting are the task number, the address space number, the address of the faulting instruction, the *EFLAGS* register and the error code which the CPU has provided upon entry into the fault handler.

If a process was running, the fault handler terminates it (in lines 282–287). Since we have not talked about processes yet, you need not worry about the `mutex_lock` and `mutex_unlock` commands (which protect the thread table) as well as the other references to the thread table via `thread_table[current_task]` or `remove_from_ready_queue()`.

We will explain all these functions and the thread table data structure later (in the Ulix book), and we will also show what the `syscall_exit()` function does. You can choose to ignore lines 282–287 in the code.

A page fault need not be a problem: it often occurs because the code attempted to access an invalid address (which is bad), but yet more often the address will be valid, but the page won't be in the physical RAM. That situation can be helped. The Ulix book describes the implementation of the *page fault handler*. It requires a working hard disk since we will *page out* pages to the disk and later *page them in* again.

page fault handler

A
Code Listings

Listing A.1: `irq.h`

```
/*
Copyright (c) 2008-2015 Felix Freiling, University of Erlangen-Nürnberg, Germany
Copyright (c) 2011-2015 Hans-Georg Eßer, University of Erlangen-Nürnberg, Germany

This program is free software: you can redistribute it and/or modify it under
the terms of the GNU General Public License as published by the Free Software
Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.  See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with
this program.  If not, see <http://www.gnu.org/licenses/>.                  */

// this is irq.h


// IRQ numbers 0 to 7 are handled by the "master PIC", numbers 8 to 15 are
// handled by the "slave PIC".
// The slave PIC generates an IRQ number 2 on the master PIC.

#define IRQ_TIMER      0
#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 gdtsel    : 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 gdtsel:  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 gdtsel, 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].gdtsel    = gdtsel;           // 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 content_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
          (always)
215
216   handler = interrupt_handlers[number];
217   if (handler != NULL)  handler (r);
218 }
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:
```

```
29  _sys_stack:
30
31  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
32
33  section .setup
34  align 4
35
36  mboot:
37          MB_HEADER_MAGIC    equ    0x1BADB002
38          ; Header flags: page align (bit 0), memory info (bit 1)
39          MB_HEADER_FLAGS    equ    11b           ; Bits: 1, 0
40          MB_CHECKSUM        equ    -(MB_HEADER_MAGIC + MB_HEADER_FLAGS)
41
42          ; GRUB Multiboot header, boot signature
43          dd      MB_HEADER_MAGIC               ; 00..03: magic string
44          dd      MB_HEADER_FLAGS               ; 04..07: flags
45          dd      MB_CHECKSUM                   ; 08..11: checksum
46
47  trickgdt:
48          dw      gdt_end - gdt_data - 1        ; GDT size
49          dd      gdt_data                      ; linear address of GDT
50
51  gdt_data:
52          ; selector 0x00: empty entry
53          dd      0, 0
54
55          ; code selector 0x08:
56          ; base 0x40000000, limit 0xFFFFFFFF, type 10011010, flags 1100
57          db      0xFF, 0xFF, 0, 0, 0, 10011010b, 11001111b, 0x40
58
59          ; data selector 0x10:
60          ; base 0x40000000, limit 0xFFFFFFFF, type 10010010, flags 1100
61          db      0xFF, 0xFF, 0, 0, 0, 10010010b, 11001111b, 0x40
62  gdt_end:
63
64  global start
65  start:
66          ; BEGIN higher half trick
67          lgdt    [trickgdt]
68          mov     ax, 0x10
69          mov     ds, ax
70          mov     es, ax
71          mov     fs, ax
72          mov     gs, ax
73          mov     ss, ax
74          jmp     0x08:higherhalf   ; far jump to the higher half kernel
75
76  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
77
```

```
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: reenable 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;
```

```
}


// syscall_handler() is the generic system call handler which
// is called when a process executes the "int 0x80" instruction.

void
syscall_handler (context_t *r) {
  void (*handler) (context_t*);   // handler is a function pointer
  int number = r->eax;
  if (number != __NR_get_errno) set_errno (0); // default: no error
  handler = syscall_table[number];
  if (handler != 0)  handler (r);
  else
    printf ("Unknown syscall no. eax=0x%x; ebx=0x%x. eip=0x%x, esp=0x%x. "
            "Continuing.\n", r->eax, r->ebx, r->eip, r->esp);
}



// handle errno

int
get_errno () {
  if (scheduler_is_active)  return thread_table[current_task].error;
  else                      return startup_errno;
}

void
set_errno (int err) {
  if (scheduler_is_active)  thread_table[current_task].error = err;
  else                      startup_errno = err;
}



// setup syscall handlers

void
install_all_syscall_handlers () {
  install_syscall_handler (__NR_brk, syscall_sbrk);
  install_syscall_handler (__NR_get_errno, syscall_get_errno);
  install_syscall_handler (__NR_set_errno, syscall_set_errno);
  install_syscall_handler (__NR_fork, syscall_fork);
  install_syscall_handler (__NR_exit, syscall_exit);
  install_syscall_handler (__NR_waitpid, syscall_waitpid);
  install_syscall_handler (__NR_resign, syscall_resign);
  install_syscall_handler (__NR_gettid, syscall_gettid);
  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[pid].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
              memory
328         if (pd->ptds[i].present) {           // directory entry in use
329           page_table *pt = (page_table*) (PHYSICAL ((pd->ptds[i].frame_addr) <<
                12));
330           for (int j = 0;  j < 1024;  j++) {
331             if (pt->pds[j].present) {        // table entry in use
332               int pageno = i*1024 + j;
333               int index = hash ((as << 20) | pageno, PG_MAX_COUNTERS);
334               if (pick_as==-1 && counter_table[index].used) {
335                 // initialize minimum, pick
336                 pick_as     = as;
337                 pick_pageno = pageno;
338                 themin = counter_table[index].count;
339               } else {
340                 if (counter_table[index].count < themin) {
341                   themin = counter_table[index].count;
342                   pick_as     = as;
343                   pick_pageno = pageno;
344                 }
345               }
346             }
347           }
348         }
349       }
350     }
351   }
352   mutex_unlock (paging_lock);
353
354   if (pick_as != -1) {
355     mutex_lock (paging_lock);
356     page_out (pick_as, pick_pageno);
357     mutex_unlock (paging_lock);
358   } else {
359     printf ("\nERROR: cannot pick a page to evict!\n");
360   }
361 }
362
363 void
364 syscall_clrscr (context_t *r) {
365   // no parameters, no return value
366   vt_clrscr ();
367 }
```

```
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
        0..31
591     set_errno (EINVAL);
592     retval = -1; goto end;
593   }
594
595   // check if current process may send a signal
596   if ((thread_table[current_task].euid == 0) ||
597       (thread_table[target_pid].euid == thread_table[current_task].euid)) {
598     retval = u_kill (target_pid, signo);
599   } else {
600     set_errno (EPERM);
601     retval = -1;
602   }
603   end: r->eax = retval;
604
605   // run scheduler if this was a raise operation
606   if (current_task == target_pid) {
607     asm (".intel_syntax noprefix; \
608       mov eax, 66;; \
609       int 0x80;; \
610       .att_syntax; ");
611   }
```

```
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   }
```

Index

K

L

M

N

O

P

R

S

T

Z

Bibliography

[EF15] Hans-Georg Eßer and Felix C. Freiling. *The Design and Implementation of the Ulix Operating System.* 2015. To be published.

[Fri05] Brandon Friesen. Bran's Kernel Development. A tutorial on writing kernels. Version 1.0, 2005. http://www.osdever.net/bkerndev/Docs/title.htm; accessed 2013/10/19.

[Int86] Intel. Intel 80386 Programmer's Reference Manual, 1986. http://microsym.com/editor/assets/386intel.pdf, accessed 2013/11/17.

[Int88] Intel. Intel 8259A Programmable Interrupt Controller (8259A/8259A-2), December 1988. Datasheet for PC interrupt controller chip, http://i30www.ira.uka.de/~sdi/doc/8259a.pdf.

[OSD13] OSDev.org. Interrupt Descriptor Table, July 2013. http://wiki.osdev.org/Interrupt_Descriptor_Table, last accessed 2013/11/16.