Writing an Operating System

Jonas Freiknecht

www.jofre.de

October 2010 – April 2011

Version 0.3

BETA: This book is still in a beta status, meaning it is not done. At the moment I try to find a way to implement a working paging mechanism which is a little bit more flexible than the one in the "Step10beta" folder. This is the reason for me to publish this tutorial at the current state. If you have an idea how to get paging running (Including a heap and memory allocation algorithm feel free to contact me. You could be the reason for me to keep writing ☺).

Content

Content 2

I Writing an Operating System 4

I.I Introduction 4

I.II Goal of this tutorial 4

I.III What do I need? 5

I.IV Structure of this tutorial 6

I.V Compiling 7

I.VI What is an operating system? 7

I.VII Copyright 8

1 The boot loader 8

2 The entry to our OS 11

2.1 Entering the world of C 13

3 Writing on screen 15

4 Let me interrupt you 21

5.1 Interrupt Descriptor Table 21

4.2 Interrupts to handle exceptions 23

4.3 Interrupt Requests to handle hardware messages 32

5 Our first input device – the keyboard 37

6 The Programmable Interval Timer 39

7 Sounds 42

8 Administrating memory with the Global Descriptor Table 43

8.1 Task State Segment 43

8.2 Local Descriptor Table 44

8.3 Call Gate 44

9 Extended Output 47

10 Paging 52

10.1 Page Entry 52

10.2 Page faults 53

10.3 Adding a debugging function to extio.c 54

Literature 55

I Writing an Operating System

October, 2010
www.jofre.de

I.I Introduction

I read a lot of stuff on the internet about Operating systems and came to the fantastic idea to write my own one. This is a little fictitious you might think. Of course it is! But although I was sure that it will be a hard piece of work I started making a structure and defined the working packages that will be included in the final – let us call it OS (Operating System) from now on – OS.

One very good reason to convince me starting this task is that I – as many of you, too, I guess – learn by doing. As you will see I explain a chapter or a function and try to put the idea into code. So you can directly understand what I just explained. In my eyes this is the way that enables us all to learn easily and quickly.

And now I have to pull the break. Of course, we will not write the perfect OS that is comparable to Windows or Linux. We will write a very short version that – in the end – works and shows the general aspects of such software.

I promise in this book I will concentrate on the most important facts and I will leave out long explanations and historical facts. Whoever is interested in more details I recommend the book "Operating Systems – Design and Implementation" by Tannenbaum and Woodhull which is the Bible of all OS books.

In this text you will often find hints which are a nice to have but are not directly belonging to the topic of operating systems[1].

Now, let us define what we will learn in this tutorial and what our OS is supposed to be able to.

I.II Goal of this tutorial

In this tutorial we will write a small operating system which should help you to learn the basic parts:

- Boot loader – How do we tell our BIOS (basic input/output system) to boot our OS?
- Kernel – Why is the kernel called "core" of an OS? How can we access and control resources such as memory or processors?
    - System calls – How can we enable programmers to use these resources easily and keep them away from accessing our hardware directly?
    - Process management – Which process gets how much time of the CPU (central processing unit) and how do processes communicate between each other?

[1] But I did spend time writing them so invest the few minutes and read them! :-)

- File systems – How to write to hard disk / floppy disk? How do we know which space on the disk is free and which area is in use? What happens when high language procedures like “write” are used?
- The shell – How do we communicate with our OS and how are commands parsed? How do I specify files or other processes as input / output of a process?
- Devices – How do I access hardware such as my graphic card?

Last but not least, you will learn to compile the code we write which is not very easy! I will show you how which tools we use and how we combine our commands to a batch file.

Hint: All tools we use are free!

I.III What do I need?

I have made the decision that use Windows to write the OS. If you are using Linux do not panic! Equal tools are available under Linux, too. All you need is an assembler, a C compiler, a virtual machine, GRUB (Grand Unified Bootloader) and some disk tools.

- **Assembler** - The entrance code to our kernel is written in assembler and – unfortunately – cannot be written in C[2]. I decided to work with NASM (Netwide Assembler) which can be downloaded for free here: http://www.nasm.us/

Hint: There are different assemblers which are not compatible between each other. Whoever tried to compile a GAS-Assembler file with NASM will know that the expressions in the language differ a lot. For this tutorial I have chosen NASM because it is easy to read and well documented.

- **C compiler** - The rest of our code is written in C. This code will be compiled using DJGPP which can be downloaded here: http://www.delorie.com/djgpp/zip-picker.html

Hint: Installing DJGPP is a little tricky because there is no installer that let you choose the needed components. You have to download each package on its one and unzip it into a single directory.

Hint: If you are installing DJGPP pay attention that you add the necessary paths[3] of the bin directory in your system path dir. This enables your BAT-file to find the applications we are using without entering entire paths. The same holds for NASM!

- **The Virtual Machine** - As a Virtual Machine you can take probably every Tool you like. I have chosen Virtual PC because it is easy to use and I believe it is a little faster than for instance VM Ware and does not create some weird network connections that slow down your system. Furthermore, it is able to create Virtual Floppy Disk Images.
- **RawWrite** - Helps you to copy images to that we create later onto a (virtual) floppy disc. This tool can be found here: http://www.chrysocome.net/rawwrite

[2] Defining our stack as well as defining interrupts is easier in assembler if not impossible in C.

[3] http://cse.iitkgp.ac.in/pds/software/gcc/DJGPP.html

- **Virtual Floppy Drive** - If you have no floppy drive of if you want to work with a virtual machine you will need to download VFD (Virtual Floppy Drive): http://sourceforge.net/projects/vfd/

- **GRUB4DOS** - This is the simple boot manager to load our kernel. It is very easy to use and we do not need to write an own boot loader! http://download.gna.org/grub4dos/

- **Build Floppy Image** - A simple tool that can create floppy images that can be written on (virtual) floppy disks: http://www.nu2.nu/bfi/

- A good text editor, I propose Notepad++ which has code highlighting for nearly every language that is available on earth. http://notepad-plus-plus.org/

- **Coffee**

As you can see we are using a lot of different tools (8!) but be aware that you will need some of them only when it comes to compiling. Most time you will spend with Notepad++ :-)

I.IV Structure of this tutorial

In the following paragraphs I will show you the code that is newly added to the OS. At the end I will tell you the lines that are added to our compilation batch file so that you can test our output immediately. The code of each chapter can be found in an archive that is called like the chapter. This helps you to avoid typos in your code or if you want to skip a chapter you can easily take the code from the current archive. I will add a lot of comments to the code so that I try to avoid explanations afterwards.

Technically seen we will move from left to right as seen in illustration 1. First, we will see how a boot loader works and create a dynamic image that we only need to copy our kernel on. Afterwards, we will create the entry point to the kernel in assembler. This assembler code now calls a main function that you might now from a standard C program. Now that our kernel is running and waiting for orders we can create other programs that can be run on our system. These programs use system calls to use the resources of the hardware. System calls are a set of functions we will offer to programmers to build an abstraction layer between hardware and software.

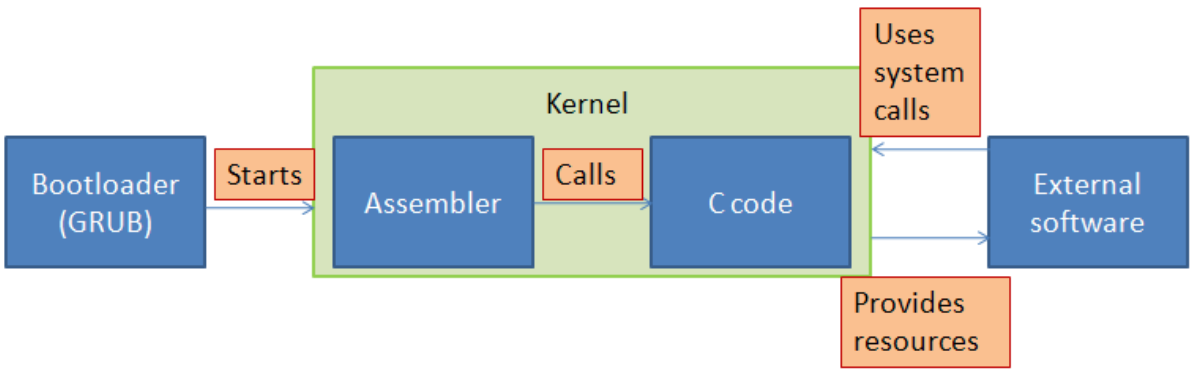

Image 1: Structure of the Operation System

I.V Compiling

In my eyes, compiling is a tricky part, especially under Windows because Linux has many tool preinstalled when it comes to compiling and creating images. When you have installed all the tools mentioned above it is time to create a batch file that compiles our code.

```
@echo off
echo Assembling...
nasm -f aout -o start.o start.asm
echo Done!

echo Compiling...
rem -> More to come here
echo Done!

echo .. and Linking!
 ld -T link.ld -o kernel.bin start.o
echo Done!

pause
```

Listing 1: Compilation Script „build.bat“

As you can see we process our compilation in 4 steps.

1) Assembling our assembly code which is basically the entry point to our OS. This code cannot be written in C this is why we have to use a (rather short) piece of assembler code.

2) Compiling the C code. This will be done by DJGPP's GCC. For each source and header file we add we have to add a line to the batch file.

3) Linking the object files. The compiler produces so called object files with the extension .o. These files will have to be linked to a binary file, in our case "kernel.bin". This will be done by DJGPP's linker called "ld". Similar to step 2 each object file that is created by the GCC has to be added to the linker.

4) Finally we will have to copy the "kernel.bin" to a floppy disk that is prepared with our boot manager GRUB. This step is explained in detail in chapter XX.

Hint: If the batch file tells you that either "nasm", "gcc" or "ld" is an unrecognized command you will have to check if you added the necessary directories to your class path AND rebooted your system.

I.VI What is an operating system?

An OS can be characterized by two abilities. It extends our computer by separating hardware from software and it manages its resources.

- **An OS separates the Hardware Layer from the Application Layer.** When a programmer writes a tool like the Notepad he does not want to care about free memory, free RAM,

determining and writing to free hard disc sectors. These tasks will be done by the OS by so called[4] **system calls which extend our machine.**

- **An OS manages resources.** Computers consist of processors, memories, timers, disks, mice, network interfaces, and printers etc. Tasks of an Operating System are to manage the allocation of processors, memory and IO devices.

But what about process management, file management, a user system and the shell? Of course, these are all parts of an OS but each topic that you can image can be put under the two topics above.

I.VII Copyright

1 The boot loader

Each operating system needs an entry point that is recognized by the bios and says “Hey, I can provide a bootable binary!”. This entry point is called boot loader and has a very typical structure so that it is recognized as such:

- It is (at least) 512 bytes long
- It ends with the signature 055h, 0AAh on the last two bytes

Hint: A boot loader can theoretically be bigger than 512 bytes. The most important thing is that the bytes at position 511 (055h) and position 512 (0AAh) are correct.

To train our assembler skill we could write this piece of code on our own but on the one hand this is a tricky process because you can easily make mistakes in assembler and on the other hand there is an easy to apply out of the box boot loader that is, furthermore, free to use: The Grand Unified Bootloader (GRUB).

Hint: If you are interested in a professionally written boot loader have a look at the one of the free and open source Minix OS. There even exists a tutorial on how its boot process works in detail[5].

Maybe you already heard about GRUB and want to know how this fine software works. This is easy, we will create a floppy disk image, put copy a file from the GRUB archive on it, create a simple configuration file, use a tool to make our image bootable and copy our binary file that contains our kernel on it.

This image can then be written on a (virtual) floppy disk and when we then start our computer we will see a nice boot screen which let us pick the OS we want to boot.

To build a boot image we need two tools:

---

[4] In chapter XX we will learn more about system calls.

[5] http://www.os-forum.com/minix/boot/

- BFI (Build floppy image)
- GRUB4DOS

Let us create the image:

1) Create a folder named “floppy” and put the file “grldr” from the GRUB archive in it.
2) Execute BFI with the following parameters:
   bfi -t=144 -f=floppy.img floppy
   This creates us a 1.44 MB image called floppy.img and takes the folder “floppy” as source.
3) Now we install GRUB on the image and make it bootable:
   bootlace.com --fat12 --floppy floppy.img
4) Write the “floppy.img” onto a (virtual) floppy disk by using RawWrite. Now open the disk and create a new file called “menu.lst”. This is where GRUB reads the operating systems from when it is booted.
5) Write these three lines in “menu.lst”:
   title My first OS
   kernel /kernel.bin
   boot
   The title is the string that is shown in the OS selection dialog, kernel is the absolute path to the kernel binary that we will start to create in chapter 1 and boot tells GRUB that he should boot the kernel when our OS is selected.

I will include the “floppy.img” in the archive so that you can use it.

Next, start Virtual Floppy Drive (VFD) and start the driver.

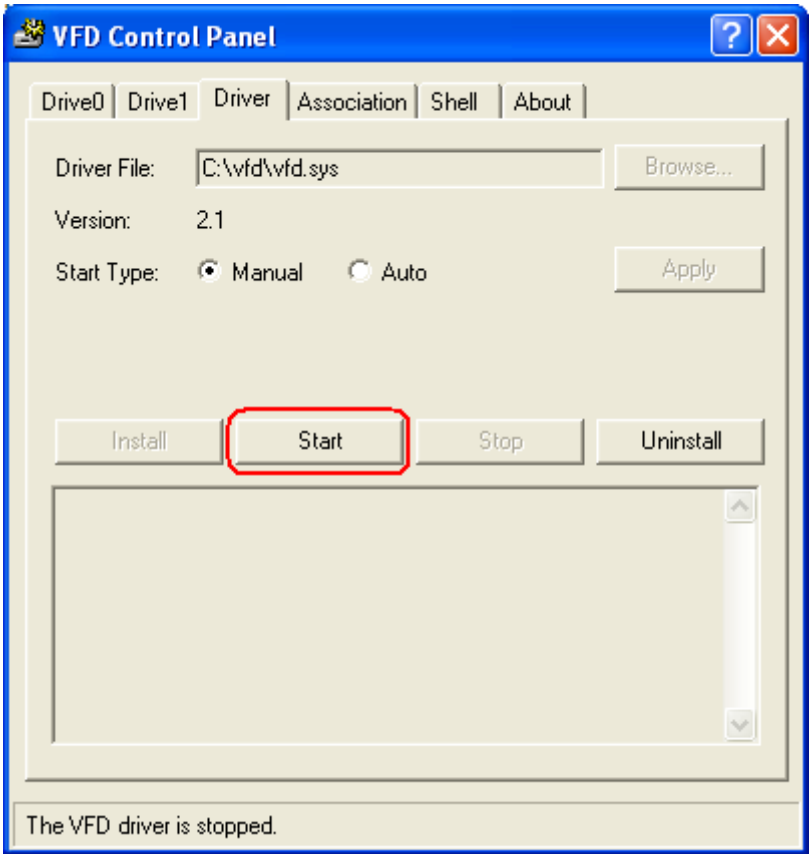

Image 2: Starting the VFD driver

Then you select the „Drive0“ tab and open a virtual floppy disk which can be found in each “step” folder (but is always the same). Assign a drive letter on top of the screen. I will use A: in this tutorial.

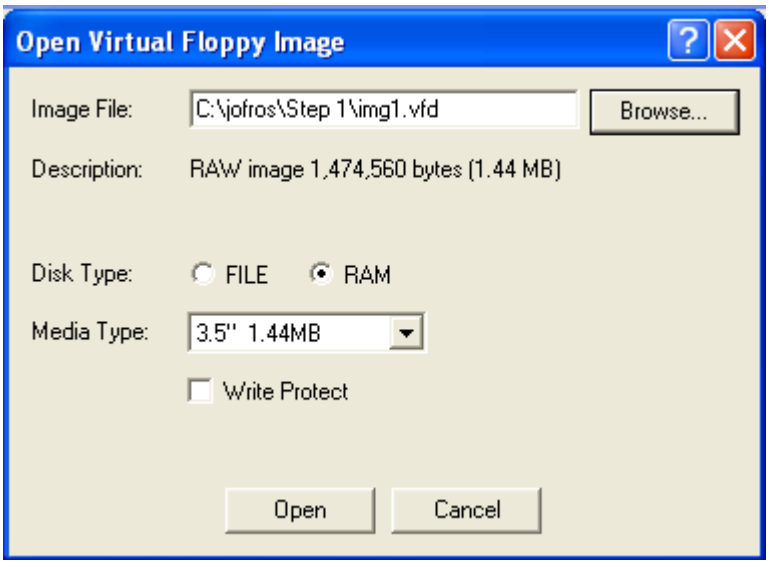

Image 3: Open a virtual floppy disk

You will see that you have a working floppy drive that you can use now. If you want you can close VFD. Once the driver is running it will not stop until you reboot your system.

Now, you can start VirtualPC. Click „Floppy Disk“ in the main menu, select „Control physical drive A:“. The Virtual Machine will recognize you virtual drive „A:“ as own drive so that you can boot from it.

Hint: When you want to use the „img1.vfd“ in VirtualPC instead of a virtual floppy drive you need to start „VFD Control Panel“, select the tab „Drive0“ and „Save…“ the content of your drive „A:“ to the „img1.vfd“.

Hint: You will also have to create a virtual system. Therefore, a HDD size of 64 mb and a RAM size of 64 mb is more than sufficient.

Illustration 4 shows what your screen should look like when you boot from your floppy disk.

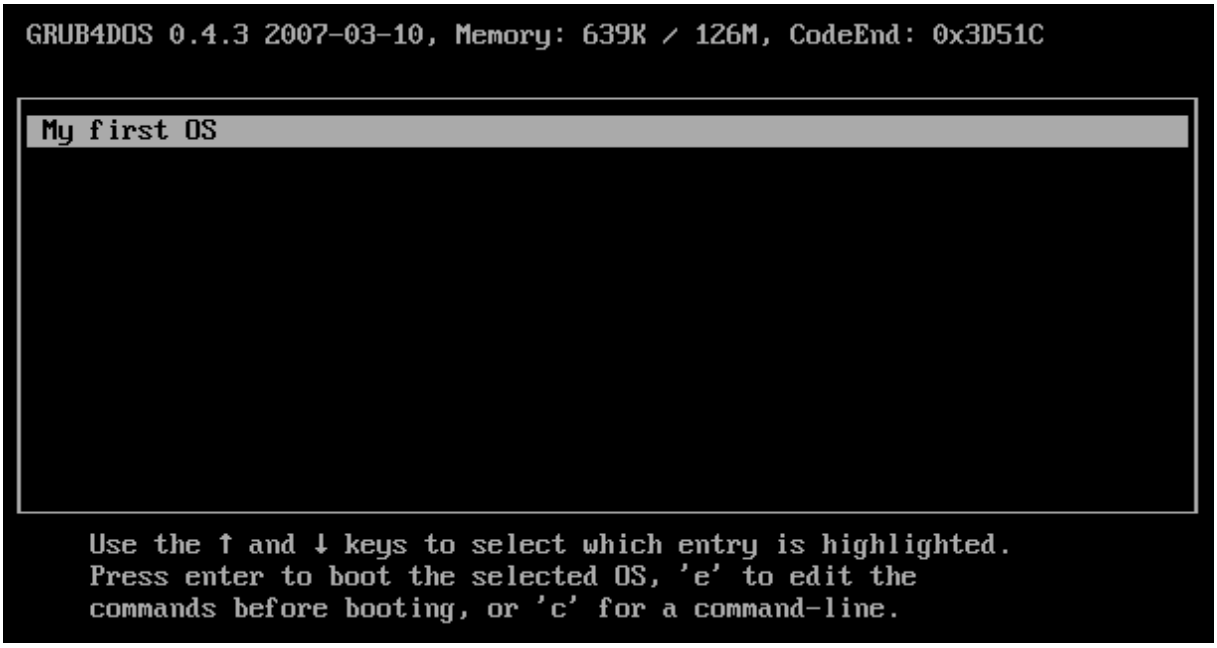

Image 4: GRUB showing our kernel

2 The entry to our OS

Now we finally write code!

```
; This is our entry point. As mentioned in this tutorial we write this
code in assembler
; since it is easier to define our stack here and to mark this file as
bootable binary for
; GRUB.

; We are moving in a 32 bits environment.
[BITS 32]
global start
start:
    mov esp, _sys_stack     ; This points the stack to our new stack area
    jmp stublet

; This part MUST be 4byte aligned so that GRUB can read the magic number
(boot signature),
; the flags and the checksum.
ALIGN 4

; This part is to make our kernel GRUB compatible and tell grub some
flags, for
; instance that we use the AOUT binary format instead of ELF. This is
neccessary
; because we use DJGPP under Windows.
```

```
mboot:
    ; Multiboot macros to make a few lines later more readable
    MULTIBOOT_PAGE_ALIGN      equ 1<<0
    MULTIBOOT_MEMORY_INFO     equ 1<<1
    MULTIBOOT_AOUT_KLUDGE     equ 1<<16
    MULTIBOOT_HEADER_MAGIC    equ 0x1BADB002
    MULTIBOOT_HEADER_FLAGS    equ MULTIBOOT_PAGE_ALIGN |
MULTIBOOT_MEMORY_INFO | MULTIBOOT_AOUT_KLUDGE
    MULTIBOOT_CHECKSUM  equ -(MULTIBOOT_HEADER_MAGIC +
MULTIBOOT_HEADER_FLAGS)
      ; Extern tells the compiler that the label can be found in another
module.
    EXTERN code, bss, end

    dd MULTIBOOT_HEADER_MAGIC
    dd MULTIBOOT_HEADER_FLAGS
    dd MULTIBOOT_CHECKSUM

    ; When we build our kernel the assembler fills these values
automatically.
      ; For mboot the address of the "mboot" label is set. This is to show
GRUB
      ; where it can find which section.
    dd mboot
    dd code
    dd bss
    dd end
    dd start

; Everything is defined properly. Now we can start our main loop in
assembler.
stublet:
    jmp $


; Place holder for GDT code.


; Place holder for interrupt code.


; The bss section holds data that is not jet defined. For example, a "dd"
; would be ignored by the compiler.
; Now we define our stack of 8KB. Remember that a stack actually grows
; downwards, so we declare the size of the data before declaring
; the identifier '_sys_stack'
SECTION .bss
    resb 8192 ;reserve 8KB
_sys_stack:
```

Listing 2: The kernel called "start.asm"

What we do here is easy:

1) We save a pointer to our stack (What is a stack?[6]) in the ESP register.

2) We define a lot of macros to enable multi booting for our OS in GRUB.

[6] A stack is a region in our memory where we can store values of registers, variables and other objects. A stack can be accessed via push (write) and pop (read). More on stacks can be found here: http://en.wikipedia.org/wiki/Stack_%28data_structure%29

3) We define an endless loop that keeps our OS alive.

4) We define our stack.

What does our OS do so far? Nothing :-) But if you compile the code from the folder “Step 1” by clicking the batch file “build.bat” and following the last 2 steps from the chapter “Compiling” you will see that the image is recognized as Operating System and you can boot from a disk if you try.

2.1 Entering the world of C

For now, this is all the assembler code we need. The next parts can be realized using C code. Later, when it comes to interrupts we will have to switch back to assembler again.

What we do now is we declare an extern function “_main” and call it. Therefore, add the following two lines beneath the line “stublet:”.

```
stublet:
    extern _main
    call _main
    jmp $
```

Listing 3: Calling our main method

Hint: It is not necessary to call the entry function to our C code “main”. I have only chosen this name because C programmers might be used to this name and see it as entry point to a program and so to our OS.

Now that we have called the function we have to define it! Create a new file “main.c” and fill it with the following functions.

Hint: The underscore in “_main” is there because the compiler adds a “_” to every function and variable declaration. This is to avoid naming conflicts.

```
#include <system.h>

// copy count bytes from src to dest
void *memcpy(void *dest, const void *src, size_t count)
{
    const char *sp = (const char *)src;
    char *dp = (char *)dest;
    for(; count != 0; count--) *dp++ = *sp++;
    return dest;
}

// set count bytes in dest to val
void *memset(void *dest, char val, size_t count)
{
    char *temp = (char *)dest;
    for( ; count != 0; count--) *temp++ = val;
    return dest;
}

// Same as above, but this time, we're working with a 16-bit 'val' and
dest pointer.
unsigned short *memsetw(unsigned short *dest, unsigned short val, size_t
count)
{
    unsigned short *temp = (unsigned short *)dest;
```

```
    for( ; count != 0; count--) *temp++ = val;
    return dest;
}

// Returns the length of a character array.
size_t strlen(const char *str)
{
    size_t retval;
    for(retval = 0; *str != '\0'; str++) retval++;
    return retval;
}

// Reading from IO ports
unsigned char inportb (unsigned short _port)
{
    unsigned char rv;
    __asm__ __volatile__ ("inb %1, %0" : "=a" (rv) : "dN" (_port));
    return rv;
}

// Writing to IO ports
void outportb (unsigned short _port, unsigned char _data)
{
    __asm__ __volatile__ ("outb %1, %0" : : "dN" (_port), "a" (_data));
}

// The main function loops for eternity. This is to keep our OS alive.
int main()
{
    for (;;);
    return(0);
}
```

Listing 4: Our first C file "main.c"

Well, here does not jet happen a lot. From our “start.asm” we call “main” which ends up in an endless loop. The functions we defined are very basic functions to set memory and read and write to ports. To make them available in all coming source files we need to define the function headers in a header file. We create a “system.h” in a subdirectory called “Include”.

```
#ifndef __SYSTEM_H
#define __SYSTEM_H

/* MAIN.C */
extern void *memcpy(void *dest, const void *src, size_t count);
extern void *memset(void *dest, char val, size_t count);
extern unsigned short *memsetw(unsigned short *dest, unsigned short val,
size_t count);
extern size_t strlen(const char *str);
extern unsigned char inportb (unsigned short _port);
extern void outportb (unsigned short _port, unsigned char _data);

#endif
```

Listing 5: Header file "system.h"

Hint: If you are not firm with the language C you will want to know what header files are. Header files tell the compiler which functions will later on be defined in our source code and what they will look

like. The *"#ifndef … #define … #endif*" construct makes sure that each header file is only included once[7].

In this second step we succeeded in calling a c function in our OS! Many of you will get really exited now because they think "Hey, now I can do anything a can do under my preferred OS!". Actually, this is wrong. As you can see we defined functions like *memcpy*, a very basic function in C. All other functions we will have to write on our own now. Even printing on the screen is not possible so far using the functions we have at the moment. But this will change in the next chapter.

Now we have to extend our compiler script! Add the following line directly after "echo Compiling…".

```
gcc -Wall -O -fstrength-reduce -fomit-frame-pointer -finline-functions -
fno-builtin -I./Include -c -o main.o main.c
```

Listing 6: Add a C / H file to the "build.bat" script

Some of you might recognize gcc as a C compiler. Well, we need it because we are compiling C code now.

The parameters we use are:

- Wall:
- -o: "gcc" produces object files which are linked by the linker script "ld" after compiling.
- fstrength-reduce:
- fomit-frame-pointer:
- finline-functions:
- fno-builtin:
- I: Tells "gcc" to browse the following directory for header files. We will pass the path to our "include" directory.

If you start the "build.bat" from folder "Step 2" you will see … nothing again :-) Frustrating, isn't it? But after all the compiler tells us (by not complaining) that our code is free of errors and that we are going in the right direction!

3 Writing on screen

Now it is time to write something that you can show your friends and family! We will finally print something on the screen now that we have a basic structure.

We want to set us two targets for this chapter. On the one hand print colored characters on the screen and on the other hand scrolling our text whenever we want to. The VGA video card your computer should own makes it pretty simple. The only thing we have to do is to put the character and their color in a special memory area. Then we call a print function and that's it! The VGA card will take care of updating the screen with the given properties. Scrolling is a thing we have to take care of on our own. If you want to you might consider calling it creating a driver what we are doing next.

[7] More on header files and defines can be read here:

The address space I just talked about can be found at 0xB8000 in our physical memory. The buffer has the data type short with a size of 16 bits. This 16 bit element can be separated 2 times. The first 8 bits contain the character that is written on the screen and the last 8 bits contain the foreground and the background color.

Value	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	Light Grey
8	Dark grey
9	Light blue
10	Light green
11	Light cyan
12	Light red
13	Light magenta
14	Light brown
15	White

Table 1: Color values for the VGA video card

And now we print our line on the screen like:

Println(„My first text\nis pretty red!“,4,1);

Unfortunately, we do not. The reason is simple. We are moving on a very low level and have not yet defined any functions to easily print out text. We have to see the screen as a matrix (80x25) of characters and colors (our 16 bit values) that is provided by the VGA video card. This means we have to access a matrix that can hold 25 lines with 80 colored characters which is aligned in a linear buffer.

Do you know how to break down a two dimensional environment down to a one dimensional (or linear)? There exists a simple equation:

index = (y_value * width_of_screen) + x_value;

So when we want to write a character to the destination (3,4) (Third character in the fourth line) we have to calculate (4*80)+3 which results in 323.

```
unsigned short *where = (unsigned short *)0xB8000 + 323;
*where = character | (attribute << 8);
```

Listing 7: Determining a character on screen and changing it

With this knowledge we are able to draw colored text on our screen. In the „scrn.c“ I included the „system.h“ because we need functions to copy memory, determining the length of strings and writing to I/O ports (in this case of the VGA card).

The scroll function is easy. It copies the line 1 over the line 0, line 2 over line 1, … and clears the last line.

```
#include <system.h>

// Local variables for text pointer, background and foreground color
// and for the cursor coordinates
unsigned short *textmemptr;
int attrib = 0x0F;
int csr_x = 0, csr_y = 0;

// Function to scroll the screen for one line if needed.
void scroll(void)
{
    unsigned blank, temp;

    // We define a blank line.
    blank = 0x20 | (attrib << 8);

    // If we find ourselves at the last line we need to scoll.
    if(csr_y >= 25)
    {
        // We move the entire text on the screen up one line.
        temp = csr_y - 25 + 1;
        memcpy (textmemptr, textmemptr + temp * 80, (25 - temp) * 80 * 2);

        // We set the last line to our blank line.
        memsetw (textmemptr + (25 - temp) * 80, blank, 80);
        csr_y = 25 - 1;
    }
}

// Updates the blinking cursor. This is done by the VGA adapter
// as you can see on the functions "outportb" where we write
// to an external IO Port.
void move_csr(void)
{
    unsigned curpos;

    // Find the position where the cursor has to be placed.
    curpos = csr_y * 80 + csr_x;

      // Here we write the position to the Control Register of
      // the VGA controller.
    outportb(0x3D4, 14);
    outportb(0x3D5, curpos >> 8);
    outportb(0x3D4, 15);
    outportb(0x3D5, curpos);
```

```
}

// Clear the screen
void cls()
{
    unsigned blank;
    int i;

    // Like the scrolling function we define a blank line.
    blank = 0x20 | (attrib << 8);

    // Now we put this blank line in all 25 lines of our screen.
    for(i = 0; i < 25; i++)
        memsetw (textmemptr + i * 80, blank, 80);

    // Set the cursor to 0/0 and set the blinking cursor there.
    csr_x = 0;
    csr_y = 0;
    move_csr();
}

// Deletes the last char
void delchar()
{
      unsigned short *where;
      unsigned att = attrib << 8;
      char c = ' ';

      csr_x --;

      if(csr_x < 0)
      {
            csr_x = 0;
            csr_y --;

            if(csr_y < 0)
                  csr_y = 0;
      }

      where = textmemptr + (csr_y * 80 + csr_x);
      *where = c | att;
      move_csr();
}

// Print a char on the screen.
void putch(unsigned char c)
{
    unsigned short *where;
    unsigned att = attrib << 8;

    // Backspace -> Move the cursor back
    if(c == 0x08)
    {
        if(csr_x != 0) csr_x--;
    }
      // Moves the cursor foreward to a point that can be divided by 8
    else if(c == 0x09)
    {
        csr_x = (csr_x + 8) & ~(8 - 1);
    }
      // Carriage Return -> Brings the cursor back to the beginning
      // of the current line
```

```
    else if(c == '\r')
    {
        csr_x = 0;
    }
    // New line -> increments the y value and sets the x value to 0.
    else if(c == '\n')
    {
        csr_x = 0;
        csr_y++;
    }
    // Everything else greater / equal than a space can be printed out.
      // Character and color will be printed out.
    else if(c >= ' ')
    {
        where = textmemptr + (csr_y * 80 + csr_x);
        *where = c | att;
        csr_x++;
    }

    // If we have printed out more than 79 characters
      // we add a new line.
    if(csr_x >= 80)
    {
        csr_x = 0;
        csr_y++;
    }

    // Scroll the screen and move the cursor.
    scroll();
    move_csr();
}

// Puts out a string which is now easy using putch
void puts(unsigned char *text)
{
    int i;

    for (i = 0; i < strlen(text); i++)
    {
        putch(text[i]);
    }
}

// Sets the foreground and background color
void settextcolor(unsigned char forecolor, unsigned char backcolor)
{
      // First 4 bytes for background, last 4 bytes for foreground.
    attrib = (backcolor << 4) | (forecolor & 0x0F);
}

// Initialize our video driver
void init_video(void)
{
      // @0xB8000 our video memory begins that holds the information
      // on the content of our screen.
    textmemptr = (unsigned short *)0xB8000;
    cls();
}
```

Listing 8: Controlling the screen via "scrn.c"

That is a bunch of code! Let me give you a first insight in what the functions do:

1) **Void scroll(void):** Scrolls the screen by one line.
2) **void move_csr(void):** Updates the little blinking cursor which is always at the position where we are currently typing.
3) **void cls():** Clears the entire screen.
4) **void delchar():** Deletes the last character and resets the cursor.
5) **void putch(unsigned char c):** Puts one single character on the screen.
6) **void puts(unsigned char *text):** Puts an entire string on the screen.
7) **void settextcolor(unsigned char forecolor, unsigned char backcolor):** Sets the foreground and background color.
8) **void init_video(void):** Initializes our video functions.

As you might have seen we use functions that we declared in main.c. If you have problems understanding the purpose of them you can now figure out what they are used for.

Hint: If you want to know more about VGA programming have a look at this page: http://www.brackeen.com/vga/basics.html

At last, we have to add the prototypes of these functions to our "system.h" file. Add these lines directly before "#endif".

```
/* SCRN.C */
extern void cls();
extern void putch(unsigned char c);
extern void puts(unsigned char *str);
extern void settextcolor(unsigned char forecolor, unsigned char
backcolor);
extern void init_video();
```

Listing 9: Prototypes for the screen functions in "system.h"

Hint: You will see that I add all function prototypes and structs to "system.h". This is no good programming style. Normally you would create an own header file for each code file. But I personally prefer to keep the overview over our written functions and at the end of this tutorial we will have more than XX code files so that you will get bored searching for a function prototype in all the header files.

Now, open your „main.c" and add the following 2 lines at the beginning of the „main" function.

```
init_video();
puts("Hello World!");
```

Listing 10: Printing on screen

Now try it. Copy the compiled „kernel.bin" to the GRUB floppy disk and boot it. This is the first time you will see something. This is the most complex Hello World you have ever written am I right?

4 Let me interrupt you

After output comes? Right, input! But before we start to ask ourselves how we can use the keyboard we have to talk about another topic first: Interrupts. What is an interrupt? An interrupt can be compared to a trigger for the processor. Every time, a routine wants the processors attention it creates an interrupt, for example when it finishes a command like writing data to a disk, when it has data to be read and when an input device has data that wants to be processed. Such as our keyboard later on! But an interrupt can also handle exceptions like, for instance, division by zero.

So we can say interrupts can be used for two purposes:

- Process hardware messages
- Process exceptions

A motherboard contains some chips (The PITS - we will talk about them later) for interrupt handling which are programmable so that we can define our own interrupts. These definitions are stored in the so called Interrupt Description Table.

5.1 Interrupt Descriptor Table

The Interrupt Descriptor Table (IDT) is there to tell the CPU what Interrupt Service Routine (ISR) to use to handle exceptions or hardware interrupts.

As the name says the IDT can be imagined as a table with n 64 bits long entries. It has an entry for an address where the CPU can find the ISR that is called when the interrupt occurs. Furthermore, it has a permission flag which tells which permissions are necessary to call this interrupt. There are permission levels from 0 (highest permission) to 3 (lowest). Commonly, the kernel itself has the highest permission and applications that are executed on the system have the permission level 3. The same permission level technique you will find in the Global Descriptor Table when we talk about memory and task management in chapter XX.

GRAFIK DER 64 bits

The following source code will explain the IDT a lot better.

```
#include <system.h>

// An entry entry in the IDT table
struct idt_entry
{
    unsigned short base_lo;
    unsigned short sel;        // The kernel segment
    unsigned char always0;     // This value is always ... 0!
    unsigned char flags;
    unsigned short base_hi;
} __attribute__((packed)); // We use the smalest alignment, meaning no
zeros between
// our variables.

struct idt_ptr
{
    unsigned short limit;
```

```
    unsigned int base;
} __attribute__((packed));

// The IDT with 256 entries. We will only use 32 entries. If any other
// IDT entry is called it will cause an "Unhandled Interrupt" exception.
struct idt_entry idt[256];
struct idt_ptr idtp;

// This extern function is defined in start.asm. It initializes a new
// IDT (idtp). This can once again only be done in assembler.
extern void idt_load();

// Set an entry in the IDT.
void idt_set_gate(unsigned char num, unsigned long base, unsigned short
sel, unsigned char flags)
{
    // IR's base address
    idt[num].base_lo = (base & 0xFFFF);
    idt[num].base_hi = (base >> 16) & 0xFFFF;

      // Other properties are set here
    idt[num].sel = sel;
    idt[num].always0 = 0;
    idt[num].flags = flags;
}

// Initialize the IDT
void idt_install()
{
    // Sets the max address of the IDT
    idtp.limit = (sizeof (struct idt_entry) * 256) - 1;
      // Sets the start address of the IDT
    idtp.base = &idt;

    // Set the entire IDT to zero
    memset(&idt, 0, sizeof(struct idt_entry) * 256);

      // Here you can add new ISRs to the IDT via idt_set_gate

    // Tells the Processor where the new IDT can be found
    idt_load();
}
```

Listing 11: The Interrupt Descriptor Table

So what do we do here? We create 2 structs, idt_entry and idt_ptr. The first holds an entry in the IDT to define an ISR with its start and end point in memory so that it can be called. The idt_ptr struct holds the information on the start and end in memory of the new IDT. Using the function idt_install we can now set the IDT memory to the place where an array of 256 idt_entries can be found.

Hint: The packed directive tells the compiler to avoid zeros between the fields of the struct "idt_entry". This can be useful if to save memory and to keep a certain structure in your code so that an instance of the struct can be read by other procedures which expect exactly this structure. This is what our CPU does. It expects the "idt_entry" to have an exact structure (short, short, char, char, short) so that it can be processes by the CPUs routines.

We initialize this memory with zero and tell via an external function (idt_load) in our start.asm where the processor can find our new IDT which is our idt struct. Add the following code after the line "; Place holder for interrupt code.".

```
; Loads the IDT from idt.c as new IDT
global _idt_load
extern _idtp
_idt_load:
    lidt [_idtp]
    ret
```

Listing 12: Loading the new IDT in start.asm

Using the remaining function idt_set_gate we can define new ISRs. Be aware that we have no checking for valid values here! Let's expect our OS to work properly :-)

Last but not least add these three lines to system.h.

```
/* IDT.C */
extern void idt_set_gate(unsigned char num, unsigned long base, unsigned
short sel, unsigned char flags);
extern void idt_install();
```

Listing 13: Making our IDT functions public in system.h

4.2 Interrupts to handle exceptions

Can you remember that interrupts handle exceptions as well? As next step we will define the most important exceptions for our OS. An exception could be "Division by 0" or "Debug exception" and is defined as case that is encountered when the processor cannot continue the normal code execution.

We will add 32 exceptions to our IDT:

Exception number	Description	Error Code?
0	Division By Zero Exception	No
1	Debug Exception	No
2	Non Maskable Interrupt Exception	No
3	Breakpoint Exception	No
4	Into Detected Overflow Exception	No
5	Out Of Bounds Exception	No
6	Invalid Opcode Exception	No
7	No Coprocessor Exception	No
8	Double Fault Expcetion	Yes
9	Coprocessor Segment Overrun	Yes

	Exception	
10	Bad TSS Exception	Yes
11	Segment Not Present Exception	Yes
12	Stack Fault Exception	Yes
13	General Protection Fault Exception	Yes
14	Page Fault Exception	Yes
15	Unknown Interrupt Exception	No
16	Coprocessor Fault Exception	No
17	Alignment Check Exception (486+)	No
18	Machine Check Exception (586+ / Pentium)	No
19 – 31	Reserved Exceptions	No

Table 2: Exceptions in the IDT

Some exceptions give back an error code, meaning an error code is pushed on the stack. In our later code we will make it easier by pushing the error code 0 on the stack for all exceptions which do not give back an error code. To know which ISR has been called we will push the ID of the exception on the stack as well. If we are in an IRS we first deactivate interrupts using the assembler code “cli”. This avoids two interrupt routines to be called at the same time. In the following code we will define our interrupt routines which consist of quite a lot of code. Since we are working a lot with the stack and registers we will use assembler code again.

```
; The interrupt service routines definitions
global _isr0
global _isr1
global _isr2
global _isr3
global _isr4
global _isr5
global _isr6
global _isr7
global _isr8
global _isr9
global _isr10
global _isr11
global _isr12
global _isr13
global _isr14
global _isr15
global _isr16
global _isr17
global _isr18
global _isr19
```

```
global _isr20
global _isr21
global _isr22
global _isr23
global _isr24
global _isr25
global _isr26
global _isr27
global _isr28
global _isr29
global _isr30
global _isr31

;  0: Divide By Zero Exception
_isr0:
    cli
    push byte 0
    push byte 0
    jmp isr_common_stub

;  1: Debug Exception
_isr1:
    cli
    push byte 0
    push byte 1
    jmp isr_common_stub

;  2: Non Maskable Interrupt Exception
_isr2:
    cli
    push byte 0
    push byte 2
    jmp isr_common_stub

;  3: Int 3 Exception
_isr3:
    cli
    push byte 0
    push byte 3
    jmp isr_common_stub

;  4: INTO Exception
_isr4:
    cli
    push byte 0
    push byte 4
    jmp isr_common_stub

;  5: Out of Bounds Exception
_isr5:
    cli
    push byte 0
    push byte 5
    jmp isr_common_stub

;  6: Invalid Opcode Exception
_isr6:
    cli
    push byte 0
    push byte 6
    jmp isr_common_stub
```

```
;  7: Coprocessor Not Available Exception
_isr7:
    cli
    push byte 0
    push byte 7
    jmp isr_common_stub

;  8: Double Fault Exception (With Error Code!)
_isr8:
    cli
    push byte 8
    jmp isr_common_stub

;  9: Coprocessor Segment Overrun Exception
_isr9:
    cli
    push byte 0
    push byte 9
    jmp isr_common_stub

; 10: Bad TSS Exception (With Error Code!)
_isr10:
    cli
    push byte 10
    jmp isr_common_stub

; 11: Segment Not Present Exception (With Error Code!)
_isr11:
    cli
    push byte 11
    jmp isr_common_stub

; 12: Stack Fault Exception (With Error Code!)
_isr12:
    cli
    push byte 12
    jmp isr_common_stub

; 13: General Protection Fault Exception (With Error Code!)
_isr13:
    cli
    push byte 13
    jmp isr_common_stub

; 14: Page Fault Exception (With Error Code!)
_isr14:
    cli
    push byte 14
    jmp isr_common_stub

; 15: Reserved Exception
_isr15:
    cli
    push byte 0
    push byte 15
    jmp isr_common_stub

; 16: Floating Point Exception
_isr16:
    cli
    push byte 0
    push byte 16
```

```
    jmp isr_common_stub

; 17: Alignment Check Exception
_isr17:
    cli
    push byte 0
    push byte 17
    jmp isr_common_stub

; 18: Machine Check Exception
_isr18:
    cli
    push byte 0
    push byte 18
    jmp isr_common_stub

; 19: Reserved
_isr19:
    cli
    push byte 0
    push byte 19
    jmp isr_common_stub

; 20: Reserved
_isr20:
    cli
    push byte 0
    push byte 20
    jmp isr_common_stub

; 21: Reserved
_isr21:
    cli
    push byte 0
    push byte 21
    jmp isr_common_stub

; 22: Reserved
_isr22:
    cli
    push byte 0
    push byte 22
    jmp isr_common_stub

; 23: Reserved
_isr23:
    cli
    push byte 0
    push byte 23
    jmp isr_common_stub

; 24: Reserved
_isr24:
    cli
    push byte 0
    push byte 24
    jmp isr_common_stub

; 25: Reserved
_isr25:
    cli
    push byte 0
```

```
    push byte 25
    jmp isr_common_stub

; 26: Reserved
_isr26:
    cli
    push byte 0
    push byte 26
    jmp isr_common_stub

; 27: Reserved
_isr27:
    cli
    push byte 0
    push byte 27
    jmp isr_common_stub

; 28: Reserved
_isr28:
    cli
    push byte 0
    push byte 28
    jmp isr_common_stub

; 29: Reserved
_isr29:
    cli
    push byte 0
    push byte 29
    jmp isr_common_stub

; 30: Reserved
_isr30:
    cli
    push byte 0
    push byte 30
    jmp isr_common_stub

; 31: Reserved
_isr31:
    cli
    push byte 0
    push byte 31
    jmp isr_common_stub


; In isr_common_stub we will use a C function called "fault_handler"
extern _fault_handler

; Here we save the processor state, calls the C fault handler
; and restores the stack frame in the end.
isr_common_stub:
    pusha
    push ds
    push es
    push fs
    push gs
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
```

```
    mov eax, esp
    push eax
    mov eax, _fault_handler
    call eax
    pop eax
    pop gs
    pop fs
    pop es
    pop ds
    popa
    add esp, 8
    iret
```

Listing 14: Add these lines below the "_idt_load" part in start.asm

The comments in the code are self explanatory for the functionality. We basically offer interrupt routines to be called. If they are called we push the error code on the stack, save the processor state, handle the fault and restore the state on the stack.

Hint: When you ever plan to write a boot loader for your OS do not forget to turn off your interrupts when it comes to creating a stack. You can use the assembler function “cli” to disable interrupts. “sti” enables them again.

Now we have to register our ISRs to the IDT and print out an exception when the fault handler is called. By changing the text color to red we let the thing look a little bit fancier. Therefore, we create a new file called “isrs.c”.

```
#include <system.h>

// The exception handlers in start.asm
extern void isr0();
extern void isr1();
extern void isr2();
extern void isr3();
extern void isr4();
extern void isr5();
extern void isr6();
extern void isr7();
extern void isr8();
extern void isr9();
extern void isr10();
extern void isr11();
extern void isr12();
extern void isr13();
extern void isr14();
extern void isr15();
extern void isr16();
extern void isr17();
extern void isr18();
extern void isr19();
extern void isr20();
extern void isr21();
extern void isr22();
extern void isr23();
extern void isr24();
extern void isr25();
extern void isr26();
extern void isr27();
extern void isr28();
```

```
extern void isr29();
extern void isr30();
extern void isr31();

// Here we register the first 32 ISRs in our IDT. The access flag is
// set to 0x8E which means the entry is present and running in ring 0
(kernel mode)
// and has the lower bytes set to the required '14'.
void isrs_install()
{
    idt_set_gate(0, (unsigned)isr0, 0x08, 0x8E);
    idt_set_gate(1, (unsigned)isr1, 0x08, 0x8E);
    idt_set_gate(2, (unsigned)isr2, 0x08, 0x8E);
    idt_set_gate(3, (unsigned)isr3, 0x08, 0x8E);
    idt_set_gate(4, (unsigned)isr4, 0x08, 0x8E);
    idt_set_gate(5, (unsigned)isr5, 0x08, 0x8E);
    idt_set_gate(6, (unsigned)isr6, 0x08, 0x8E);
    idt_set_gate(7, (unsigned)isr7, 0x08, 0x8E);

    idt_set_gate(8, (unsigned)isr8, 0x08, 0x8E);
    idt_set_gate(9, (unsigned)isr9, 0x08, 0x8E);
    idt_set_gate(10, (unsigned)isr10, 0x08, 0x8E);
    idt_set_gate(11, (unsigned)isr11, 0x08, 0x8E);
    idt_set_gate(12, (unsigned)isr12, 0x08, 0x8E);
    idt_set_gate(13, (unsigned)isr13, 0x08, 0x8E);
    idt_set_gate(14, (unsigned)isr14, 0x08, 0x8E);
    idt_set_gate(15, (unsigned)isr15, 0x08, 0x8E);

    idt_set_gate(16, (unsigned)isr16, 0x08, 0x8E);
    idt_set_gate(17, (unsigned)isr17, 0x08, 0x8E);
    idt_set_gate(18, (unsigned)isr18, 0x08, 0x8E);
    idt_set_gate(19, (unsigned)isr19, 0x08, 0x8E);
    idt_set_gate(20, (unsigned)isr20, 0x08, 0x8E);
    idt_set_gate(21, (unsigned)isr21, 0x08, 0x8E);
    idt_set_gate(22, (unsigned)isr22, 0x08, 0x8E);
    idt_set_gate(23, (unsigned)isr23, 0x08, 0x8E);

    idt_set_gate(24, (unsigned)isr24, 0x08, 0x8E);
    idt_set_gate(25, (unsigned)isr25, 0x08, 0x8E);
    idt_set_gate(26, (unsigned)isr26, 0x08, 0x8E);
    idt_set_gate(27, (unsigned)isr27, 0x08, 0x8E);
    idt_set_gate(28, (unsigned)isr28, 0x08, 0x8E);
    idt_set_gate(29, (unsigned)isr29, 0x08, 0x8E);
    idt_set_gate(30, (unsigned)isr30, 0x08, 0x8E);
    idt_set_gate(31, (unsigned)isr31, 0x08, 0x8E);
}

// This string array contains the message corresponding
// to the exceptions.
unsigned char *exception_messages[] =
{
    "Division By Zero",
    "Debug",
    "Non Maskable Interrupt",
    "Breakpoint",
    "Into Detected Overflow",
    "Out of Bounds",
    "Invalid Opcode",
    "No Coprocessor",

    "Double Fault",
    "Coprocessor Segment Overrun",
```

```
    "Bad TSS",
    "Segment Not Present",
    "Stack Fault",
    "General Protection Fault",
    "Page Fault",
    "Unknown Interrupt",

    "Coprocessor Fault",
    "Alignment Check",
    "Machine Check",
    "Reserved",
    "Reserved",
    "Reserved",
    "Reserved",
    "Reserved",

    "Reserved",
    "Reserved",
    "Reserved",
    "Reserved",
    "Reserved",
    "Reserved",
    "Reserved",
    "Reserved"
};

// This fault handler is used in every ISR. The parameter tells which
exception
// happened. If the interrupt is valid (id < 32) we print the exception
message
// and halt the system by an endless loop.
void fault_handler(struct regs *r)
{
    if (r->int_no < 32)
    {
        settextcolor(4,0);
        puts(exception_messages[r->int_no]);
        puts(" Exception. System Halted!\n");
        for (;;);
        settextcolor(15,0);
    }
}
```

Listing 15: Create "isrs.c" to register the ISRs and to put out fault messages

In the “fault_handler” procedure we use a struct “regs” which represents a stack frame and lets us take a snapshot to handle multiple interrupts. Add this struct to “system.h”. Furthermore, add the prototype of “isrs_install”.

```
// Here we can save our stack
struct regs
{
    unsigned int gs, fs, es, ds;
    unsigned int edi, esi, ebp, esp, ebx, edx, ecx, eax;
    unsigned int int_no, err_code;
    unsigned int eip, cs, eflags, useresp, ss;
};

/* ISRS.C */
extern void isrs_install();
```

Listing 16: The regs struct to take a stack snapshot and the ISR prototype

Now we can call “idt_install();” and “isrs_install();” in our main.c. And here it gets interesting! Add the line “putch(10 / 0);” before the endless loop in the main routine. If you did everything right you will see a red line with the error code we defined for the exception “Division by zero”.

4.3 Interrupt Requests to handle hardware messages

Interrupt Requests are interrupts that are issued by hardware. Whenever a CPU receives an interrupt request (IRQ) it pauses whatever it is doing and executes the necessary action like reading from the keyboard. Afterwards it writes the hex value 0x20 to a command register of the Programmable Interrupt Controller (PIC) to tell that he has finished its action. So what is a PIC? A PIC is a chip on the motherboard to manage IRQs. Each motherboard has two of them and each PIC can handle 8 IRQs. The second PIC can also be told that the CPU finished a command by writing the value 0xA0 to the command register.

So what is the advantage of a *Programmable* Interrupt Controller? Easy! Normally, IRQ0 – IRQ7 are mapped to the IDT entry 8 to 15 and IRQ8 – IRQ15 are mapped to IDT entry 112 – 120. As you can remember we reserved IDT entry 0-31 for exceptions! So we will remap IRQ0-IRQ15 to the IDT entries 32 – 47. Again, we will extend our start.asm. Add the following lines after the block “isr_common_stub” (after the line “iret”).

```
; Handling Hardware Interrupt Requests
global _irq0
global _irq1
global _irq2
global _irq3
global _irq4
global _irq5
global _irq6
global _irq7
global _irq8
global _irq9
global _irq10
global _irq11
global _irq12
global _irq13
global _irq14
global _irq15

; 32: IRQ0
_irq0:
    cli
    push byte 0
    push byte 32
    jmp irq_common_stub

; 33: IRQ1
_irq1:
    cli
    push byte 0
    push byte 33
    jmp irq_common_stub

; 34: IRQ2
_irq2:
    cli
```

```
    push byte 0
    push byte 34
    jmp irq_common_stub

; 35: IRQ3
_irq3:
    cli
    push byte 0
    push byte 35
    jmp irq_common_stub

; 36: IRQ4
_irq4:
    cli
    push byte 0
    push byte 36
    jmp irq_common_stub

; 37: IRQ5
_irq5:
    cli
    push byte 0
    push byte 37
    jmp irq_common_stub

; 38: IRQ6
_irq6:
    cli
    push byte 0
    push byte 38
    jmp irq_common_stub

; 39: IRQ7
_irq7:
    cli
    push byte 0
    push byte 39
    jmp irq_common_stub

; 40: IRQ8
_irq8:
    cli
    push byte 0
    push byte 40
    jmp irq_common_stub

; 41: IRQ9
_irq9:
    cli
    push byte 0
    push byte 41
    jmp irq_common_stub

; 42: IRQ10
_irq10:
    cli
    push byte 0
    push byte 42
    jmp irq_common_stub

; 43: IRQ11
_irq11:
```

```
    cli
    push byte 0
    push byte 43
    jmp irq_common_stub

; 44: IRQ12
_irq12:
    cli
    push byte 0
    push byte 44
    jmp irq_common_stub

; 45: IRQ13
_irq13:
    cli
    push byte 0
    push byte 45
    jmp irq_common_stub

; 46: IRQ14
_irq14:
    cli
    push byte 0
    push byte 46
    jmp irq_common_stub

; 47: IRQ15
_irq15:
    cli
    push byte 0
    push byte 47
    jmp irq_common_stub

extern _irq_handler

irq_common_stub:
    pusha
    push ds
    push es
    push fs
    push gs

    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov eax, esp

    push eax
    mov eax, _irq_handler
    call eax
    pop eax

    pop gs
    pop fs
    pop es
    pop ds
    popa
    add esp, 8
    iret
```

Listing 17: Remapping the Interrupt Service Requests

Once again, we react on interrupts. When, for example, the Interrupt Request 15 occurs interrupts will be forbidden, we push a dummy error code 0 on the stack followed by the index of the interrupt to match it to the entry in the Interrupt Descriptor Table afterwards. Then we save our registers to the stack, handle the Interrupt Request and restore our registers from the stack again.

As you might guess the C part follows now. Create a new file and call it “irq.c”.

```
#include <system.h>

// We define these Interrupt Service Requests on our own
// to point to a special IRQ handler instead of the regular
// fault_handler.
extern void irq0();
extern void irq1();
extern void irq2();
extern void irq3();
extern void irq4();
extern void irq5();
extern void irq6();
extern void irq7();
extern void irq8();
extern void irq9();
extern void irq10();
extern void irq11();
extern void irq12();
extern void irq13();
extern void irq14();
extern void irq15();

// Pointer array to handle custom ORQ handlers of a special IRQ
void *irq_routines[16] =
{
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0
};

// With this function we can define a custom IRQ handler
// for an IRQ. So we manage the target functions that are called
// when an interrupt occurs on our own.
void irq_install_handler(int irq, void (*handler)(struct regs *r))
{
    irq_routines[irq] = handler;
}

// By setting the pointer of an interrupt routine to 0
// we will unload the handler.
void irq_uninstall_handler(int irq)
{
    irq_routines[irq] = 0;
}

// Here we remap our IRQs (0-15 to 32-47) as explained in the
// tutorial.
void irq_remap(void)
{
    outportb(0x20, 0x11);
    outportb(0xA0, 0x11);
    outportb(0x21, 0x20);
    outportb(0xA1, 0x28);
```

```
    outportb(0x21, 0x04);
    outportb(0xA1, 0x02);
    outportb(0x21, 0x01);
    outportb(0xA1, 0x01);
    outportb(0x21, 0x0);
    outportb(0xA1, 0x0);
}

// Identical to defining the exception handlers we will
// install the appropriate Interrupt Service Routines to the
// corresponding entries in the IDT.
void irq_install()
{
    irq_remap();

    idt_set_gate(32, (unsigned)irq0, 0x08, 0x8E);
    idt_set_gate(33, (unsigned)irq1, 0x08, 0x8E);
    idt_set_gate(34, (unsigned)irq2, 0x08, 0x8E);
    idt_set_gate(35, (unsigned)irq3, 0x08, 0x8E);
    idt_set_gate(36, (unsigned)irq4, 0x08, 0x8E);
    idt_set_gate(37, (unsigned)irq5, 0x08, 0x8E);
    idt_set_gate(38, (unsigned)irq6, 0x08, 0x8E);
    idt_set_gate(39, (unsigned)irq7, 0x08, 0x8E);

    idt_set_gate(40, (unsigned)irq8, 0x08, 0x8E);
    idt_set_gate(41, (unsigned)irq9, 0x08, 0x8E);
    idt_set_gate(42, (unsigned)irq10, 0x08, 0x8E);
    idt_set_gate(43, (unsigned)irq11, 0x08, 0x8E);
    idt_set_gate(44, (unsigned)irq12, 0x08, 0x8E);
    idt_set_gate(45, (unsigned)irq13, 0x08, 0x8E);
    idt_set_gate(46, (unsigned)irq14, 0x08, 0x8E);
    idt_set_gate(47, (unsigned)irq15, 0x08, 0x8E);
}

// Each service routine of an Interrupt Request points to this
// function. After handling the ISR we need to tell the interrupt
// controllers that we are done handling the interrupt. As said
// in the tutorial this happens by putting the hex value 0x20 to
// the adress 0x20 for the first controller and for the second
// controller we write 0x20 to 0xA0.
// If the second controller (IRQ from 8 to 15) receives and interrupt
// we need to tell the first controller, too, that we have finished
// the interrupt routine.
void irq_handler(struct regs *r)
{
      // Blank function pointer
    void (*handler)(struct regs *r);

    // If we have a routine defined for this interrupt call it!
    handler = irq_routines[r->int_no - 32];
    if (handler)
    {
        handler(r);
    }

      // Here we are already done with the handling of our interrupt!

      // If the entry in the IDT that we called had an index >= 40
      // we will have to send an "End of Interrupt" (0x20) to the
      // second controller.
    if (r->int_no >= 40)
    {
```

```
        outportb(0xA0, 0x20);
    }

      // In both cases we need to tell the first controller that
      // we are done handling our Interrupt Routine.
    outportb(0x20, 0x20);
}
```

Listing 18: "irq.c" to handle our (non exception) hardware interrupts

The source code is easy to understand and very similar to listing XX so I will not comment on it. Now we are almost done! The next step will be to add the function prototypes to “system.h”.

```
/* IRQ.C */
extern void irq_install_handler(int irq, void (*handler)(struct regs *r));
extern void irq_uninstall_handler(int irq);
extern void irq_install();
```

Listing 19: Function prototypes for the Interrupt Requests

So, it is time to use the code we have written in the last hour. Let us add the necessary initialization to our “main.c”. Add these two lines right before “for(;;);”. The first function initializes our Interrupt Requests and the last line is an assembler call to allow interrupts from now on!

```
    irq_install();
    __asm__ __volatile__ ("sti");
```

Listing 20: Initialize Interrupt Requests and allow interrupts from now on

Wow, this was a bunch of difficult code! What we finally did is we wrote procedures to let our kernel handle exceptions that occur and furthermore we are now able to react on hardware inputs! Well, not yet because we have to install our handler for the keyboard. So let us do this now!

5 Our first input device – the keyboard

Motivational Speech: We will slow down a little bit from now on. The last chapters were really difficult and now that you managed you get here proves 3 things. You are smart. You have what is called in today’s business environment “drive to achieve”. And you have passed a line that most hobby OS developers did not pass this is why you can call yourself an advanced beginner from now on :-)

Do you remember that we wrote a simple driver for the VGA controller? We will now do the same for the keyboard which is mostly converting scan codes to ASCII (Abkürzung) characters that we can understand.

Hint: Why use scan codes? As you know English keyboards look different to German ones. So when we press the “z” key on an English keyboard it will write a “y” in an OS that is set to German language. When we set the OS to English language it will print the character “z”. Even if we use the same key! This is possible because the OS maps the scan code of every key (which is an integer) to a character.

We will use a lookup table to map the scan codes to ASCII which is a simple array. Before we define this array we will have to learn about a special case. If the top bit of the byte we will receive from the

keyboard via our interrupt is set this means that the key has been released. You can check this easily with "*scancode & 0x80*".

Create a "kb.c" and add the following code:

```
#include <system.h>

// KBDUS stands for US keyboard layout.
// Create a layout in a different language if you want
// or confuse your friends with a weired mapping :-)
unsigned char kbdus[128] =
{
    0,  27, '1', '2', '3', '4', '5', '6', '7', '8',  /* 9 */
  '9', '0', '-', '=', '\b',   /* Backspace */
  '\t',                 /* Tab */
  'q', 'w', 'e', 'r',   /* 19 */
  't', 'y', 'u', 'i', 'o', 'p', '[', ']', '\n',      /* Enter key */
    0,                  /* 29   - Control */
  'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';',  /* 39 */
 '\'', '`',   0,        /* Left shift */
 '\\', 'z', 'x', 'c', 'v', 'b', 'n',                  /* 49 */
  'm', ',', '.', '/',   0,                           /* Right shift */
  '*',
    0,      /* Alt */
  ' ',      /* Space bar */
    0,      /* Caps lock */
    0,      /* 59 - F1 key ... > */
    0,   0,   0,   0,   0,   0,   0,   0,
    0,      /* < ... F10 */
    0,      /* 69 - Num lock*/
    0,      /* Scroll Lock */
    0,      /* Home key */
    0,      /* Up Arrow */
    0,      /* Page Up */
  '-',
    0,      /* Left Arrow */
    0,
    0,      /* Right Arrow */
  '+',
    0,      /* 79 - End key*/
    0,      /* Down Arrow */
    0,      /* Page Down */
    0,      /* Insert Key */
    0,      /* Delete Key */
    0,   0,   0,
    0,      /* F11 Key */
    0,      /* F12 Key */
    0,      /* All other keys are undefined */
};

// Handler for keyboard interrupts
// We can register this one with "irq_install_handler".
void keyboard_handler(struct regs *r)
{
    unsigned char scancode;

    // Read from the keyboards data buffer
    scancode = inportb(0x60);

    // Check if the key has been released
    if (scancode & 0x80)
    {
```

```
            // Use this branch to check if shift, alt, control, ... has
been released
    }
    else
    {
            // Here a key is pressed. When you keep holding the key
            // down you will receive interrupts repeatedly.

            // Here we will output each char that we receive.
        putch(kbdus[scancode]);
    }
}

// Install the keyboard interrupt request handler
void keyboard_install()
{
    irq_install_handler(1, keyboard_handler);
}
```

Listing 21: IRQ to handle keyboard input

The motherboard has an own microcontroller for the keyboard. This controller has two channels, one for the mouse and one for the keyboard. Additionally, it has two registers: A data register (0x60) and a control register (0x64). Anything the keyboard sends to the computer is stored in the data register.

You might recognize that some values in the layout map are left at 0. We will make up our own values for these keys. Furthermore, it will be necessary to add a routine to save the state of some keys like capslock, numlock, scrolllock, alt, control, … to allow uppercase letters!

EXTEND KB.C

Add the function prototype for the keyboard_installation to the “system.h”.

```
/* KEYBOARD.C */
extern void keyboard_install();
```

And call this function in the “main.c” right before the line “__asm__ __volatile__ ("sti");”. Compile your code and see that you can write on the screen. Check the automatic scrolling that we implemented in the output part!

6 The Programmable Interval Timer

You might have heard of it as system clock. It provides three different channels. Channel 0 is bound to IRQ0, channel 1 is for system use and should never been accessed, channel 2 is for the system speaker.

We will use channel 0 to schedule the CPU times to new processes later on as well as to make the current process sleep for a certain time. By default the tick rate is set to generate 18222 interrupts per second. The reason for this is that if a tick occurs all 0.055 seconds and if we use a 16 bit timer tick counter the counter will overflow after exactly one hour and will automatically set back to 0.

Once again we use the outportb function to set the timer interval for firing IRQ0. Each of the three channels I mentioned has 3 data registers 0x40, 0x41 and 0x42 as well as a command register on

0x43. The timer of the PIT is able to change the frequency it is firing and it gets even better, we can set a timer for each of our three channels. But why do we only have one command register? The answer is we use a divisor register. In such a register the command we enter is split up in binary code and can be separated in several properties.

- CNTR – Counter of the channel (0-2)
- RW – Read Write Mode (1 = Least Significant Byte, 2 = Most Significant Byte, 3 = LSB then MSB)
- Mode – 0 = Interrupt on terminal count, 1 = Hardware retriggerable one shot, 2 = Rate generator, 3 = Square Wave Mode, 4 = Software strobe, 5 = Hardware strobe
- Binary Coded Decimal (BCD) – 0 = 16 bit counter, 1 = 4x BCD decade counter

So what does this all mean? We write a 16 bit value into the command register which is separated into CNTR, RW, Mode and BCD. The CNTR value should be clear; it defines the channel we want to change (Interrupt Timer for IRQ0, System and Speaker). The RW property tells if we want to write the first 8 bit of the 16 bits, the last 8 bits or both. Since we write the entire 16 bits of the data we want to write to the data register we pick both (our frequency will be an integer). I do not want to talk about each mode in detail because this will take some time. You can read about them here[8]. Basically they are all modes of running the counter. They define only different ways to start the counting or the way of counting.
Anyway, we will use mode 3, the square wave generator. Last but not least we have to define if we want to use a binary coded decimal. As you can guess we will not! We will transfer our 16 bit counter.
Now we have to write a hex value to the command register 0x43. How do we build this command up? Easy, we pick the counter (0), the RW mode (3), the mode (3) and the BCD (0) and form this to a binary value which is: 00 11 011 0. Now use a calculator to transform this binary value to a hexadecimal value and we will receive: 36.

The timer will divide it's input clock of 1193180Hz by the number of we put into the data register to figure out many times per second it should fire the signal for the channel.

So we are ready to write some code again! Create a new file "timer.c" and add the following lines.

```
#include <system.h>

// For how many ticks has our system been running?
int timer_ticks = 0;

// Set the timer frequency
void timer_phase(int hz)
{
    int divisor = 1193180 / hz;
    outportb(0x43, 0x36); // Write the command we assembled
                                    // in the tutorial to the command
```

[8] http://en.wikipedia.org/wiki/Intel_8253

```
register
    outportb(0x40, divisor & 0xFF); // Set the high byte of the frequency
    outportb(0x40, divisor >> 8); // Set the low byte of the frequency
}

// This one handles the interrupt for the timer. We increment
// the timer_ticks everytime the interrupt is fired. By
// default the timer ticks 18222 times per second.
void timer_handler(struct regs *r)
{
    timer_ticks++;

      // Every 18 ticks (~1 second) we can execute an action here.
    if (timer_ticks % 18 == 0)
    {
        puts("One second has passed\n");
    }
}

int nDummy = 0;
void dummy() {
      nDummy = nDummy + 1;
}

// This is a wait function which loops until a special
// time was waited for.
void timer_wait(int ticks)
{
    unsigned long eticks;

    eticks = timer_ticks + ticks;
    while(timer_ticks < eticks)
      {
            dummy();
      }
}

// We set up the system clock for IRQ0 to register to our
// timer.
void timer_install()
{
    irq_install_handler(0, timer_handler);
}
```

Listing 22: "Timer.c" to enable our system clock

You will wonder aobut the function “dummy”. This is a tricky problem I was not able to solve yet. Generally, it should be sufficient to use the line “while(timer_ticks < eticks);” in our “timer_wait” function but it seems that this loop only ends when there is a command between the brackets that actually does something. An empty dummy function is no solution. Of course, this is not very pretty but it works.

Add the following lines to the “system.h”.

```
/* TIMER.C */
extern void timer_wait(int ticks);
extern void timer_install();
```

Finally, add “timer_install();” to your main method in “main.c” and see what happens!

7 Sounds

Now that you read the chapter on the PIT, you will know why I talk about sounds now. You will remember that its channel 2 is designed to control the speaker. And now that we know about the command register of the PIT it is pretty simple to implement some easy functions to control the speaker.

Hint: Why do we still need the speaker? We can use the audio signals to tell the user of our OS about the current state of his machine. An example would be an exception that is fired through an interrupt. On the one hand we use red letters but on the other hand we can strengthen this warning by a deep tone. You will have the chance to try this later and see: it works :-)

Create a new file called “speaker.c” and add the following lines:

```
#include <system.h>

// Play sound using built in speaker
void play_sound(u32int nFrequence) {
      u32int Div;
      u8int tmp;

      //Set the PIT to the desired frequency
      Div = 1193180 / nFrequence;
      outportb(0x43, 0xb6);
      outportb(0x42, (u8int) (Div) );
      outportb(0x42, (u8int) (Div >> 8));

      //And play the sound using the PC speaker
      tmp = inportb(0x61);
      if (tmp != (tmp | 3)) {
            outportb(0x61, tmp | 3);
      }
}

 //make it shutup
void nosound() {
      u8int tmp = (inportb(0x61) & 0xFC);
      outportb(0x61, tmp);
}

 //Make a beep
void beep() {
      play_sound(1000);
      timer_wait(5);
      nosound();
}
```

Listing 23: "speaker.c" to control the speaker

The function “play sound” uses the command register to tell the PIT that we want to change the settings of the second channel, namely the speaker. Feel free to calculate “0xb6” to a decimal digit and try to understand the properties we talked about in chapter 6. Afterwards, we write our frequency to the data register of the pit. LAST PART.

“nosound” instead stops playing the sound. XX

The function “beep” uses one of our timer functions and beeps for exactly 5 ticks which is pretty short but sufficient to get the attention of the user.

Now we add these three lines to the “system.h” to make our functions available in the other parts of our OS:

```
/* SOUND.C */
extern void play_sound(u32int nFrequence);
extern void nosound();
extern void beep();
```

Listing 24: Extern declarations for the speaker

Try using “beep();” in the main function (after allowing interrupts: “__asm__ __volatile__ ("sti");”). If you like you can add a beep in the “fault_handler” function in “isrs.c”. Another idea to use the sound function is to write a row of beeps and waits with different frequencies. Here is a link that shows the frequencies to the corresponding tunes so that you can use your favorite song as starting melody for your OS. But keep in mind that a speaker tune could really be annoying :-)

Now we have written something to show your friends. Time for some serious theory again!

Hint: What are these strange types like “u32int”? I created some type definitions in “system.h” which help to avoid typing a lot as well as it helps to determine if a type is signed or not. Signed means that negative values are allowed. Unsigned only allows values >=0.

8 Administrating memory with the Global Descriptor Table

With the GDT (Global Descriptor Table) we answer another question from the beginning of this tutorial. How do we manage the memory and avoid access to occupied disk space? Well, the GDT is actually only a part of the memory management system but it is able to define which memory areas (segments[9]) are executable or data! The real memory management (which block is occupied and which one is free is told via the paging functions that we will take care about later on). Furthermore, the GDT can tell the kernel about so called segment violations that a process tries to access invalid memory. The kernel is then supposed to kill this process.

But the GDT is able to hold other information than segment descriptors and includes other OS parts as well. Namely the Task State Segment (TSS), the Local Descriptor Table (LDT) and the Call Gate.

8.1 Task State Segment

The TSS contains information on tasks running on the system:

- **Processor register states** – Very helpful if we have to switch between different tasks (multi tasking) to save and restore process states.
- **I/O Port permissions** – A bit array keeps track on all permissions a task has. When a task wants to access an IO port, for instance our VGA card as seen in chapter XX to set the blinking cursor, the permission is requested. It the value in the bit array corresponding to the VGA controller is 0 the access is granted. If it is 1, the access is denied.

[9] Since the GDT is a structure used by Intel processors, Intel also defined the term „segment“ as memory area.

- **Inner level stack pointers** – The TSS contains 6 fields to store a new stack pointer when we change the privilege level. As privilege level you could imagine to execute a program as *admin* and then change the privilege level to *user*. Of course, a user has not as much rights as an administrator. With these inner level stack pointers we can manage separate stacks. For example for our kernel and for applications.

- **Previous TSS links** – Linking TSS is once again be helpful for task switching. When we have to access another process with another privilege level we switch our TSS, fulfill our duty and then switch back to the TSS of the task we came from.

Hint: All TSS are linked in the Task Register which is a special segment and can (only) partially been accessed by programmers to read task information. You might know the Linux command "ps" which shows all running processes including process ID. This application uses the TSS to determine information on the different processes.

8.2 Local Descriptor Table

The LDT has almost the same structure as the GTD but it is process specific. This means through the LDT we are able to give each process its own memory. The GTD has to contain an entry to the LDT descriptor otherwise we will not be able to create a reference to it.

8.3 Call Gate

Call Gates are used to enable a process of a lower privilege level to execute code of a higher privilege level. This is necessary to enable programmers later on to use kernel functions such as system calls. A kernel mostly has a privilege level of 0 (the highest) and an application generally has a privilege level of 3 (lowest).

Okay, now we know what the tasks of a GDT are. But how does it work? You can imagine the GDT as a list of 64 bit entries. An entry contains the segment where the region we define starts, where it ends and the privilege level a process has to have to access this segment. There privilege levels are there to prevent the kernel to crash if an application tries to execute commands it is not supposed to. A good example is "cli" and "sti" to forbid and allow interrupts. If an application tries to access these commands in a non privileged sector an exception will be thrown.
The first entry (0) in the GDT is defined as the NULL descriptor. If a segment register is set to 0 it will cause a General Protection Fault (See IRS13).

P	DPL	DT	Type	G	D	0	A	Segment length
1	2	1	4	1	1	1	1	4 (bytes)

Image 5: Entry in the Global Descriptor Table

At first, I will define the fields in an entry:

Field	**Description**
P (Present)	Segment is present or not

DPL (XX Privilege Level)	In which ring does the process have to be to access this segment?
DT (Descriptor Type)	
Type	
G (Granularity)	0 = 1 byte, 1 = 4 kbytes
D (Operand size)	0 = 16 bit, 1 = 32 bit
0	Always 0
A (Available for system)	Is Always set to 0.
Segment length	Start and end of the segment

Table 3: GDT entry explanation

In our tutorial we will use a GDT with only three entries. The first one is the NULL descriptor, the first one points to our code segment is and the second one tells the kernel where it can find the data segment.

Create a new file called “gdt.c” and add the following code.

```
#include <system.h>

// This struct defines a GDT entry. Once again we define it
// as packed to keep the structure the CPU expects the
// entry to have. We do not want the compiler to optimize
// our code!
struct gdt_entry
{
    unsigned short limit_low;
    unsigned short base_low;
    unsigned char base_middle;
    unsigned char access;
    unsigned char granularity;
    unsigned char base_high;
} __attribute__((packed));

// The GDT pointer defines the range of our GDT in memory.
struct gdt_ptr
{
    unsigned short limit;
    unsigned int base;
} __attribute__((packed));

// The GDT with three entries and a pointer to our GDT.
struct gdt_entry gdt[3];
struct gdt_ptr gp;

// This function is placed in start.asm.
extern void gdt_flush();

// Put a descriptor in the GDT
void gdt_set_gate(int num, unsigned long base, unsigned long limit,
unsigned char access, unsigned char gran)
```

```
{
      // The address of the descriptor
    gdt[num].base_low = (base & 0xFFFF);
    gdt[num].base_middle = (base >> 16) & 0xFF;
    gdt[num].base_high = (base >> 24) & 0xFF;

    // Limits of the descriptor
    gdt[num].limit_low = (limit & 0xFFFF);
    gdt[num].granularity = ((limit >> 16) & 0x0F);

    // Granularity and access flag
    gdt[num].granularity |= (gran & 0xF0);
    gdt[num].access = access;
}

// Setup the GDT pointer and create the three entries (gates) we
// want to use:
// 1: Null descriptor (See Tutorial)
// 2: Describes the code segment (Base address is 0, limit is 4GBytes,
//          use 4 KBytes granularity, 32bit coding and a Code Segment
descriptor.
// 3: Data segment which is the same as the data segment except the access
//          flag which tells it is a data segment.
void gdt_install()
{
    /* Setup the GDT pointer and limit */
    gp.limit = (sizeof(struct gdt_entry) * 3) - 1;
    gp.base = &gdt;

    gdt_set_gate(0, 0, 0, 0, 0);
    gdt_set_gate(1, 0, 0xFFFFFFFF, 0x9A, 0xCF);
    gdt_set_gate(2, 0, 0xFFFFFFFF, 0x92, 0xCF);

      // Flush the old GDT and install the new one. This code
      // can be found in start.asm.
    gdt_flush();
}
```

Listing 25: The code of the Global Description Table in "gdt.c"

The code is pretty straight forward. We create a packed struct which contains all information on our memory segment entry. Then we create a struct which is basically a pointer to the GDT. The function “gdt_flush” will later on be placed in “start.asm” and flush the old GDT (which has been initialized by the GRUB loader) and initialize our new GDT. Then we create a function to put an entry into the GDT and finally we use an initialization function for the GDT which creates three entries:

- The NULL descriptor
- Entry for the code segment
- Entry for the data segment

Finally, the new GDT is installed. Therefore we add the following lines right behind “; Place holder for GDT code” to “start.asm”.

```
; We set up the segment registers to point to our GDT.
; This will set up our new segment registers. We need to do
; something special in order to set CS. We do what is called a
; far jump. A jump that includes a segment as well as an offset.
```

```
; This is declared in C as 'extern void gdt_flush();'
global _gdt_flush     ; Allow to call _gdt_flush from anywhere
extern _gp            ; Tells that _gp will be found outside start.asm (in
gdt.c)
_gdt_flush:
    lgdt [_gp]        ; Load the GDT with our '_gp' which is a special
pointer
                              ; The brackets say that we pass a function,
not the content

    mov ax, 0x10      ; 0x10 is the offset in the GDT to our data segment
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
    jmp 0x08:flush2   ; 0x08 is the offset to our code segment: Far jump!
flush2:
    ret               ; Get back to the C code after executing this one.
```

Listing 26: Installing the GDT in start.asm

Add the function prototypes to “system.h”.

```
/* GDT.C */
extern void gdt_set_gate(int num, unsigned long base, unsigned long limit,
unsigned char access, unsigned char gran);
extern void gdt_install();
```

Listing 27: Function prototypes for "gdt.c"

We are done so far, add “gdt_install” to our main function (as first line since this is an important one) and run the OS. You will not see anything; this is only an internal (but nevertheless an important) change.

Hint: You will see that we are still receiving output from the timer. Feel free to uncomment the line from “timer.c”:

```
puts("One second has passed\n");
```

9 Extended Output

We only wrote functions to output strings and chars which is was basic stuff to make quick progress and keep you with me! But what if we want to put out other data types like integers or floats?

To that point we did everything on our own. We did not use any libraries or external header files. This will change from that point on since I cannot explain every single piece of code! This would result in a ten thousand site tutorial and would surely get very confusing! Furthermore, I will not explain as detailed as I did before since we are producing more and more code that appears repeatedly. But no reason to panic! The constructs that serve the topic “Operating System programming” will still be well commented and explained!

Create a new file “extio.c”. This file will add two new functions to print out strings that may contain arguments.

```
// First include that we not write on our own
#include "stdarg.h"

// A temp char buffer
char _buffer[64];

// Returns the number representation of value "val" to the base "base".
static inline char *strfmtint(unsigned long val, int base)
{
	// We cannot have a base > 36
	if(base > 36)
		return "";

	// If the value is 0 the result is 0, too.
	if(val == 0)
		return "0";

	const char  *digits = "0123456789abcdefghijklmnopqrstuvwxyz";
	char        *p = _buffer + 64;

	int i = 0;
	*p = '\0';

	// Make each char of "val" fit into the base representation
	// we want. For example, a pointer (represented by a hex) has
	// a base of 16 and can therefore only be represented by 0-16 of
	// the char array.
	while(val)
	{
		*--p = digits[val % base];
		val /= base;

		i++;
	}

	return p;
}

// Takes the format char array and browses it for "%"s which
// indicates that we have to replace the following char by
// a value from the argument list.
// The final string is the "str" parameter whereas the result
// is the number of written parameters.
int vsprintf(char *str, const char *format, va_list arg)
{
	int written = 0;

	// Iterate through the format char array
	while(*format != '\0')
	{
		// If we hit a "%" which is the indicator for a replacement
		if(*format == '%')
		{
			format ++;

			switch(*format)
			{
				// Is our parameter a char?
				case 'c':
				{
					char temp = va_arg(arg, int);
					str[written] = temp;
```

```
                        }
                                break;

                        case 's':
                        {
                                // Is our parameter a string?
                                char *string = va_arg(arg, char *);

                                while(*string != '\0')
                                {
                                        str[written] = *string;

                                        written ++;
                                        string ++;
                                }

                                str[written] = ' ';
                                written --;
                        }
                                break;


                        case 'i':
                        {
                                // Is our parameter an integer?
                                unsigned long     temp = va_arg(arg, unsigned
long);
                                char              *string = strfmtint(temp,
10);


                                while(*string != '\0')
                                {
                                        str[written] = *string;

                                        written ++;
                                        string ++;
                                }

                                str[written] = ' ';
                                written --;
                        }
                                break;


                        case 'p':
                        {
                                // Is our parameter a pointer? //Return a
hexadecimal representation
                                unsigned long     temp = (unsigned
long)va_arg(arg, void *);
                                char              *string = strfmtint(temp,
16);


                                str[written] = '0';
                                written ++;

                                str[written] = 'x';
                                written ++;

                                while(*string != '\0')
                                {
                                        str[written] = *string;
```

```
                                    written ++;
                                    string ++;
                              }

                              str[written] = ' ';
                              written --;
                        }
                              break;

                        case 'x':
                        {
                              // Is our parameter a XX?
                              unsigned long     temp = va_arg(arg, unsigned
long);
                              char              *string = strfmtint(temp,
16);

                              while(*string != '\0')
                              {
                                    str[written] = *string;

                                    written ++;
                                    string ++;
                              }

                              str[written] = ' ';
                              written --;

                        }
                              break;

                        case '%':
                              // Or do we really want to write a "%"?
                              str[written] = '%';

                              break;

                        default:
                              str[written] = '%';
                              written ++;

                              str[written] = *format;
                              break;
                  }
            }
            else
            {
                  str[written] = *format;
            }


            written ++;
            format ++;
      }

      str[written] = '\0';
      // Return the numbers of written parameters
      return written;
}

// Convers a format char array to a char array
```

```
int sprintf(char *dst, const char *format, ...)
{
	// Create an argument list
	va_list param;
	// Put our arguments (format) into this argument list
	va_start(param, format);

	// Converts the format string to a string
	int written = vsprintf(dst, format, param);

	// Close the argument list
	va_end(param);

	// Return the number of parameters written
	return written;
}

// Our final function to simply put out a char array
// with arguments.
void putfs(const char *format, ...)
{
	// Once again read arguments, convert, ...
	// but this time we use "puts" to outpur the
	// string to our console.
	va_list param;
	va_start(param, format);

	char temp[1024];
	vsprintf(temp, format, param);

	puts(temp);
	va_end(param);
}
```

Listing 28: "extio.c" for advanced output

Here the c code starts to get tricky since we use routines which are rarely used when writing "normal" programs. These routines are, for example, to change the Zahlensystem of a variable in "strfmtint" or to read argument lists from a function header with "va_list", "va_start" and "va_end". Fortunately, we only have to understand three functions which serve one goal: To convert parameters that have a format like:

("My name is %s. I am %i years old.", "Jonas", "25")

To a string like:

"My name is Jonas. I am 25 years old."

Do not spend too much time on understanding each function in detail. This actually does not belong to the functionality of an OS. Nevertheless, it is rather useful to print values to check if the procedures we write work properly.

As usual, put the function header of of the "putfs" (which means "put formatted string") into our "system.h". Feel free to add the following line to our main method:

```
putfs("You are using %s in version %i.","JofreOS",1);
```

10 Paging

Now we are getting serious and start to make a plan to create kind of a file system. Paging is the first step in the right direction because it enables us to virtualize our memory and throw memory exceptions when a process tries to access memory that is locked or does not exist.

Let us create a simple application:

```
int main(char argc, char **argv)
{
  return 0;
}
```

Listing 29: A simple application

When you execute this program a start address will be set which identifies where the kernel has to begin executing the code. Let us assume this address is at 0x20120A8 which is at ~32mb in the address space. Why do you think a pc with less RAM will be able to execute this code without any problems? Right, because the memory is virtualized by a paging mechanism. If the user or the kernel tries accidently to access an unmapped part of memory the processor will raise a page fault exception (We defined it already in the chapter on interrupts). But how can the processor know about our paging routines? Fortunately, the x86 structure provides a basic functionality for memory mapping which is placed in the Memory Management Unit (MMU) and helps us to create a layer between the CPU and memory.

Paging works by splitting the virtual address space up by parts of 4 kb. Pages can then be mapped on frames.

10.1 Page Entry

Each page has a size of 4 KB as well as a descriptor which shows which frame it is mapped to. Frames and pages are aligned on 4KB boundaries (4kb = 1000bytes). The least 12 significant bits of the 32bit word are always 0 which can be useful to save information on the page/frame in this address space. This information should be illustrated by the following picture:

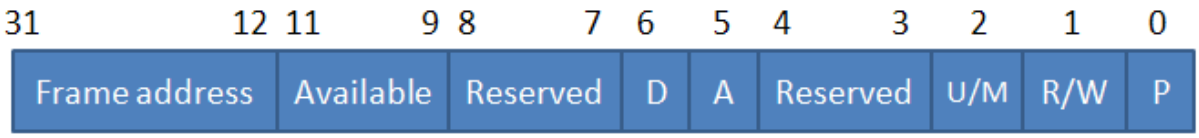

Image 6: Page Entry

- **Frame address**: As the name says the first 20 bits contain the physical address of the frame.
- **Available:** 3 unused bits which can be used by the kernel.
- **Reserved:** Internally used by the CPU.
- **D (Dirty):** The page has been written to.
- **A (Accessed):** Is set when the page has been accessed. This value is changed by the CPU.

- **Reserved:** 2 Bits which are used internally by the CPU.
- **U/M:** User mode or kernel mode page. User mode code cannot write to a kernel mode page.
- **R/W:** If this bit is set the page can be written to. If it is not set the page cannot be written to unless code that runs in kernel mode wants to write to the page.
- **P (Present):** Says if the page is present in memory.

If you have used your calculator wisely you will have figured out that a for a memory size of 4GB we need a mapping table of 4MB. The size of a table to store the page information will grow proportionally with the size of the memory. Intel came up with an intelligent idea and invented a 2 component system. The CPU knows about a so called page directory which needs, once again, 4KB. This page directory consists of n entries to page tables which might contain page table entries to describe address space. If the page tables are empty the pointer to it will be freed and the entry is being removed from the directory table by setting the present flag to 0.

10.2 Page faults

When a process accesses invalid memory areas the Memory Management Unit throws a page fault exception (ISR14) (see chapter XX). These forbidden accesses are:

1. Reading from or writing to a memory area that is not mapped.
2. A process in user mode tries to write to a read only page.
3. A process in user mode tries to access a kernel only page.
4. A page table entry is corrupted – the reserved bits have been overwritten.

Remember that some interrupts push an error code on the stack? The page fault exception is one of the exceptions that pass us information on what happened:

- Bit 0: If bit 0 is NOT set, the page was not present.
- Bit 1: If set, the operation which caused the fault was a write operation, otherwise it was a read operation.
- Bit 2: If set, the process was running in user mode, else in kernel mode.
- Bit 3: If set, the exception was raised because reserved bits were overwritten.
- Bit 4: If set, the fault occurred during an instruction fetch.

Additionally, the processor saves the address that caused the exception to the second control register.

To begin we need some memory functions which enable us to allocate memory before we can allocate memory :-) The functions we declare at the beginning serve for the heap that we will implement in the next chapter, too.

10.3 Adding a debugging function to extio.c

At the beginning we had full control over our OS. But now we implemented just another interrupt which works, like his many siblings, on his own. The PIT does nothing else but running on his own. So it is time to write a small function that holds the entire OS when an error occurs. This function we will call “puterror”. It can easily be built by functions we already defined. Add this function to “extio.c”:

```
void puterror(const char *message, const char *file, int line)
{
      // We stop our entire OS
    asm volatile("cli"); // Disable interrupts.
      settextcolor(15,4);
    putfs("Error: (\n  %s\n) at %s: %i\n",message,file,line);
    // Halt the OS
    for(;;);
}
```

Listing 30: The error function "puterror" in extio.c

A very handy feature of GCC is that its pre-compiler can determine the file name and the line number of the line it currently parses. This allows us to create the following define in “system.h”:

```
#define PUTERROR(msg) puterror(msg, __FILE__, __LINE__);
```

Listing 31: Define to automatically find source file and line number

Add "`extern void puterror(const char *message, const char *file, int line);`” to “system.h” as usual.

No you can use either “puterror” with one parameter and let the compiler find the source file and line number of the error. Or you use the function with three parameters and insert source file and line number manually.

To be continued…

Literature

http://www.jamesmolloy.co.uk/tutorial_html/index.html

https://github.com/JustSid/NANOS

http://www.ba-horb.de/~pl/BS_Skript/node1.html

http://www.osdever.net/bkerndev/Docs/title.htm

http://www.lowlevel.eu/wiki/Hauptseite

http://www.beyondlogic.org/