The little book about OS development

Erik Helin, Adam Renberg

32689bccc68ec66835d0a42fe38c5ed60407008e

Contents

Chapter 1

Introduction

What Is This Book About?

This book is not about the theory behind operating systems, or how any specific operating system (OS) works. For the first we recommend Modern Operating Systems by Andrew Tanenbaum [1]. For the second the internet is your friend.

This book is a practical guide to writing your own x86 operating system. It is designed to not give too much away with samples and code excerpts, but still give enough help with the technical details. We've tried to collect parts of the vast (and often excellent) expanse of material and tutorials out there, on the web and otherwise, and add our own insights into the problems we encountered and struggled with.

The starting chapters are naturally quite detailed and explicit, to quickly get you into coding. Later chapters give more of an outline of what is needed, as more and more of the implementation and design becomes up to the reader, who should now be more familiar with the world of kernel development. At the end of some chapters there are links for further reading, which might be interesting and give a deeper understanding of the topics covered.

The Reader

The reader of this book should be comfortable with UNIX/Linux, systems programming and the C language in general. This book is of course also a great way to get started learning those things, but it will be tougher going, especially where we implicitly make use of them. Search engines and other tutorials are often helpful.

Credits, Thanks and Acknowledgements

We'd like to thank the OSDev community [2] for their great wiki and helpful members, and James Malloy for his eminent kernel development tutorial [3]. We'd also like to thank our supervisor Torbjörn Granlund for his insightful questions and our interesting discussions.

Most of the CSS formatting of the book is based on the work by Scott Chacon for the book Pro Git, `http://progit.org/`.

Changes and Corrections

This book is hosted on Github - if you have any suggestions, comments or corrections, just fork the book, write your changes, and send us a pull request. We'll happily incorporate anything that makes this book better.

License

Chapter 2

First Steps

Developing an operating system (OS) is no easy task, and the question ” Where do I start?” is likely to come up several times during the course of the project. This chapter will help you set up your development environment and booting a very small (and boring) operating system.

Requirements

In this book, we will assume that you are familiar with the C programming language and that you have some programming experience. It will be also be helpful if you have basic understanding of assembly code.

Tools

The Easy Way

The easiest way to get all the required tools is to use Ubuntu [4] as your operating system. If you don't want to run Ubuntu natively on your computer, it works just as well running it in a virtual machine, for example in VirtualBox [5].

The packages needed can then be installed by running

```
sudo apt-get install build-essential nasm genisoimage bochs bochs-x
```

Programming Languages

The operating system will be developed using the C programming language [6][7]. The reason for using C is because developing an OS requires a very precise control of the generated code and direct access to memory, something which C enables. Other languages that provides the same features can also be used, but this book will only cover C.

The code will make use of one type attribute that is specific for GCC [8]

```
__attribute__((packed))__
```

(the meaning of the attribute will be explained later). Therefore, the example code might be hard to compile using another C compiler.

For writing assembly, we choose NASM [9] as the assembler. The reason for this is that we prefer NASM's syntax over GNU Assembler.

Shell [10] will be used as the scripting language throughout the book.

Host Operating System

All the examples assumes that the code is being compiled on a UNIX like operating system. All the code is known to compile on Ubuntu [4] versions 11.04 and 11.10.

Build System

GNU Make [11] has been used when constructing the example Makefiles, but we don't make use of any GNU Make specific instructions.

Virtual Machine

When developing an OS, it's very convenient to be able to run your code in a virtual machine instead of a physical one. Bochs [12] is an emulator for the x86 (IA-32) platform which is well suited for OS development due to its debugging features. Other popular choices are QEMU [13] and VirtualBox [5], but we will use Bochs in this book.

Booting

The first goal when starting to develop an operating system is to be able to boot it. Booting an operating system consists of transferring control along a chain of small programs, each one more "powerful" than the previous one, where the operating system is the last "program". See the following figure for an overview of the boot process.

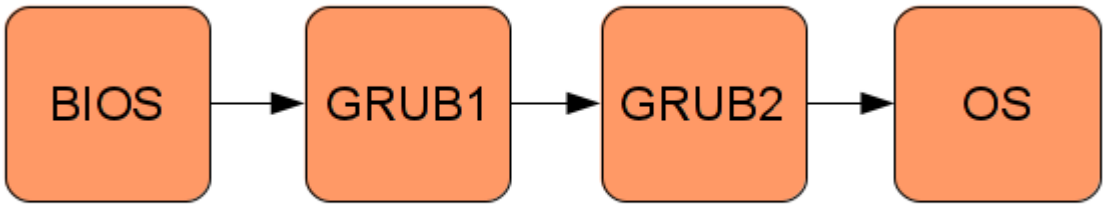

Figure 2.1: An overview of the boot process, each box is a program.

BIOS

When the PC is turned on, the computer will start a small program that adheres to the Basic Input Output System (BIOS) [14] standard. The program is usually stored on a read only memory chip on the motherboard of the PC. The original role of the BIOS program was export some library functions for printing to the screen, reading keyboard input etc. However, todays operating system does not use the BIOS functions, instead they use drivers that interacts directly with the hardware, bypassing the BIOS [14].

BIOS is a legacy technology, it operates in 16-bit mode (all x86 CPUs are backwards compatible with 16-bit mode). Today, BIOS mainly runs some early diagnostics (power-on-self-test) and then transfers control to the bootloader.

The Bootloader

The BIOS program will transfer control of the PC to a program called *bootloader*. The bootloaders task is to transfer control to us, the operating system developers, and our code. However, due to some restrictions, the bootloader is often split into two parts, so the first part of the bootloader will transfer control to the second part which will finally give the control of the PC to the operating system.

Writing a bootloader involves writing a lot of low-level code that interacts with the BIOS, so in this book, we will use an existing bootloader, the GNU GRand Unified Bootloader (GRUB) [15].

Using GRUB, the operating system can be built as an ordinary ELF [16] file, which will be loaded by GRUB into the correct memory location. However, the compilation requires some care regarding how things are to by laid out in memory, which will be discussed later in this chapter.

The Operating System

GRUB will transfer control to the operating system by jumping to the kernels start position in memory. Before the jump, GRUB will also look for a magic number to make sure that is is actually jumping to a kernel and not some random code. The magic number is part of the multiboot specification [17] which GRUB adheres to.

Hello Cafebabe

This section will show the implementation of the smallest possible operating system kernel that can be used together with GRUB. The only the kernel will do is write `0xCAFEBABE` to the `eax` register.

Compiling the Kernel

This part of the kernel has to be written in assembly, since C requires a stack, which isn't available at this point, since GRUB doesn't set one up. The code for the kernel is:

```
global loader                   ; the entry symbol for ELF

MAGIC_NUMBER equ 0x1BADB002     ; define the magic number constant
CHECKSUM     equ -MAGIC_NUMBER  ; calculate the checksum (magic number + checksum should equal 0)

section .text:                  ; start of the text (code) section
```

```
align 4                         ; the code must be 4 byte aligned
    dd MAGIC_NUMBER             ; write the magic number
    dd CHECKSUM                 ; write the checksum

loader:                         ; the loader label (defined as entry point in linker script)
    mov eax, 0xCAFEBABE         ; place the number 0xCAFEBABE in the register eax
.loop:
    jmp .loop                   ; loop forever
```

The only thing this kernel will do is write the very specific number `0xCAFEBABE` into the `eax` register. It is *very* unlikely that the number `0xCAFEBABE` would be in the `eax` register if our kernel did *not* put it there. Save the code in a file named `loader.s`.

The assembly code can now be compiled with the commando

```
nasm -f elf32 loader.s
```

to produce an 32 bits ELF [16] object file.

Linking the Kernel

The code must now be linked to produce an executable file, which requires some extra thought compared to when linking most programs. The reason for this is that we want GRUB to load the kernel at a memory address larger than or equal to `0x00100000` (1 megabyte (MB)). This is because addresses lower than 1 MB might be used by GRUB itself, by BIOS and for memory-mapped I/O. Therefore, the following linker script is needed:

```
ENTRY(loader)               /* the name of the entry label */

. = 0x00100000              /* the code should be loaded at 1 MB */

.text ALIGN (0x1000)        /* align at 4 KB */
{
    *(.text)                /* all text sections from all files */
}

.rodata ALIGN (0x1000)      /* align at 4 KB */
{
    *(.rodata*)             /* all read-only data sections from all files */
}

.data ALIGN (0x1000)        /* align at 4 KB */
{
    *(.data)                /* all data sections from all files */
}

.bss ALIGN (0x1000)         /* align at 4 KB */
{
    *(COMMON)               /* all COMMON sections from all files */
    *(.bss)                 /* all bss sections from all files */
}
```

Save the linker script into a file called `link.ld`. The executable can now be linked by running

```
ld -T link.ld -melf_i386 loader.o -o kernel.elf
```

where the file `kernel.elf` is the final executable.

Obtaining GRUB

The GRUB version we will use is GRUB Legacy, since then the ISO image can be generated on systems using both GRUB Legacy and GRUB 2. We need the GRUB Legacy `stage2_eltorito` bootloader. This file be built from GRUB 0.97 by downloading the source from `ftp://alpha.gnu.org/gnu/grub/grub-0.97.tar.gz`. However, the source code `configure` script doesn't work well with Ubuntu [18], so the binary files can be downloaded from here. Locate the file `stage2_eltorito` and copy it to your current folder.

Building an ISO Image

Now the code must be placed on a media that can be loaded by a virtual (or physical machine). In this book, we will use ISO [19] image files as the media, but one can also use floppy images, depending on what the virtual or physical machine supports.

The ISO image will be created with the program `genisoimage`. A folder must first be created that contains the files that will be on the ISO image. The following commands create the folder and copy the files to their correct places:

```
mkdir -p iso/boot/grub              # create folder structure
cp stage2_eltorito iso/boot/grub/   # copy the bootloader
cp kernel.ef iso/boot/              # copy the kernel
```

Now a configuration file `menu.lst` for GRUB must be created. This file tells GRUB where the kernel is located and configures some options:

```
default=0
timeout=0

title minios
kernel /boot/kernel.elf
```

and be placed in the `iso/boot/grub/` folder. The `iso` folder should now look like:

```
iso
|-- boot
  |-- grub
  | |-- menu.lst
  | |-- stage2_eltorito
  |-- kernel.elf
```

The ISO image can now be generated with the command

```
genisoimage -R                              \
            -b boot/grub/stage2_eltorito    \
            -no-emul-boot                   \
            -boot-load-size 4               \
            -A os                           \
            -input-charset utf8             \
            -quiet                          \
            -boot-info-table                \
            -o os.iso                       \
            iso
```

where the flags are

Flag	Description
R	Use the Rock Ridge protocol (needed by GRUB)
b	The file to boot from (relative to the root folder of the ISO)
no-emul-boot	Do not perform any disk emulation
boot-load-size	The number 512 byte sectors to load. Apparently most BIOS likes the number 4.
boot-info-table	Writes information about the ISO layout to ISO (needed by GRUB)
o	The name of the iso
A	The label of the iso
input-charset	The charset for the input files
quiet	Disable any output from genisoimage

The final ISO image `os.iso` is now available.

Running Bochs

Now that we have an ISO image with the operating system and GRUB, the final step is to run it in Bochs. Bochs needs a configuration file to start, and a simple configuration file is given below

```
megs:            32
display_library: x
romimage:        file=/usr/share/bochs/BIOS-bochs-latest
vgaromimage:     file=/usr/share/bochs/VGABIOS-lgpl-latest
ata0-master:     type=cdrom, path=minios.iso, status=inserted
boot:            cdrom
log:             bochslog.txt
mouse:           enabled=1
clock:           sync=realtime, time0=local
cpu:             count=1, ips=1000000
```

you might need to change the path to `romimage` and `vgaromimage` depending on how you installed Bochs. Save the file as `bochsrc.txt`. More information about the Bochs config file can be found at [20].

You can now start the operating system by running

```
bochs -f bochsrc.txt -q
```

`-f` tells Bochs to use the given configuration file and `-q` tells Bochs to skip the interactive start menu. You should now see Bochs starting and displaying a console with some information from GRUB on it.

After quitting Bochs, display the file log from Bochs by running

```
cat bochslog.txt
```

You should now see the state of the registers of Bochs somewhere in the output. If you find `RAX=00000000CAFEBABE` or `EAX=CAFEBABE` (depending on if you are running Bochs with or without 64 bit support), then your operating system has booted successfully!

Further Reading

- Gustavo Duertes has written an in-depth article about what actually happens when a computers boots up, `http://duartes.org/gustavo/blog/post/how-computers-boot-up`
- Gustavo then continues to describe what the kernel does in the very early stages at `http://duartes.org/gustavo/blog/post/kernel-boot-process`
- The OSDev wiki also contains a nice article about booting a computer, `http://wiki.osdev.org/Boot_Sequence`

Chapter 3

Getting to C

Now that you've managed to boot your operating system, it's time to think about the language we're currently using, assembly. Assembly is very good for interacting with the CPU and enables maximum control over every aspect of the code. However, at least for the authors, C is a much more convenient language to use. Therefore, we would like to use C as much as possible and only assembler where it make sense.

Setting Up a Stack

Before we can program in C, we need a stack. This is because we can *not* guarantee that a C program does *not* make use of a stack. Setting up a stack is not harder than to make the `esp` register point to the end of an area of free memory that is correctly aligned (remember that the stack grows towards lower addresses).

We could make `esp` just point to some area in memory, since so far, the only thing in the memory is GRUB, BIOS, the OS kernel and some memory-mapped I/O. This is not a good idea, since we don't know which address to point to, since we don't know how much memory is available, and if any memory is reserved by BIOS. A better idea is to reserve a piece of memory in the `bss` section in the ELF binary of the kernel. That way, when GRUB loads the ELF binary, GRUB will allocate memory for us.

To declare uninitialized data, the NASM pseudo-instruction `resb` [21] can be used

```
KERNEL_STACK_SIZE equ 4096                  ; size of stack in bytes

section .bss
align 4                                     ; align at 4 bytes
kernel_stack:                               ; label points to beginning of memory
    resb KERNEL_STACK_SIZE                  ; reserve stack for the kernel
```

Setting up the stack pointer is then trivial

```
mov esp, kernel_stack + KERNEL_STACK_SIZE   ; point esp to the start of the stack (end of memory area)
```

Calling C Code From Assembly

The next step is to call a C function from the assembly code. There are many different calling conventions for how to call C code from assembly [22], but we will use the *cdecl* calling convention, since it's the one

used by GCC. The cdecl calling convention states that arguments to a function should be passed via the stack (on x86). The arguments of the function should be pushed on the stack in a right-to-left order, that is, you push the rightmost argument first. The return value of the function is placed in the `eax` register. The following is an example

```
int sum_of_three(int arg1, int arg2, int arg3)
{
    return arg1 + arg2 + arg3;
}
```

```
external sum_of_three   ; the function sum_of_three is defined elsewhere

push dword 3            ; arg3
push dword 2            ; arg2
push dword 1            ; arg1
call sum_of_three       ; call the function, the result will be in eax
```

Packing Structs

In the rest of book, you will often come across "configuration bytes" that are a collection of bits in a very specific order. For example, a configuration could look like

```
Bit:     | 31           16 | 15     8 | 7     0 |
Content: | address         | index    | config  |
```

Instead of using an unsigned integer, `unsigned int`, for handling such configurations, it is much more convenient to use "packed structures". When creating the following struct in C:

```
struct example {
    unsigned char config;
    unsigned char index;
    unsigned short address;
};
```

there is no guarantee that the size of the `struct` will be exactly 32 bits, the compiler can add some padding in order to speed up element access. When using a `struct` to represent configuration bytes, it is very important that the size does *not* get padded, since the struct will eventually be treated as an unsigned integer by the hardware. To force GCC to *not* add any padding, the attribute `packed` can be used in the following way:

```
struct example {
    unsigned char config;
    unsigned char index;
    unsigned short address;
} __attribute__((packed));
```

Note that `__attribute__((packed))` is not part of the C standard, so it might not work with all compilers (it works with GCC and Clang).

Compiling C Code

When compiling the C code for the OS, quite a lot of flags to GCC has to be used. The reason for this is that the C code should *not* assume the presence of the standard library, since there is not standard library available in our OS. For more information about the flags, see the GCC manual. The flags used for compiling C are

```
-m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector -nostartfiles
-nodefaultlibs
```

As always when writing C programs, we recommend turning on all warnings and treat warnings as errors

```
-Wall -Wextra -Werror
```

You can now create a `kmain` function in a file called `kmain.c` that you can call from `loader.s`. At this point in time, `kmain` probably won't need any arguments, but in later chapters it will.

Build Tools

Now is also probably a good time to set up some build tools to make it easier to compile and run the OS. We recommend using Make [11], but there are plenty of other build systems available. A simple Makefile for the OS could look like the following

```
OBJECTS = loader.o kmain.o
CC = gcc
CFLAGS = -m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector \
         -nostartfiles -nodefaultlibs -Wall -Wextra -Werror -c
LDFLAGS = -T link.ld -melf_i386
AS = nasm
ASFLAGS = -f elf

all: kernel.elf

kernel.elf: $(OBJECTS)
    ld $(LDFLAGS) $(OBJECTS) -o kernel.elf

os.iso: kernel.elf
    cp kernel.elf iso/boot/kernel.elf
    genisoimage -R                              \
                -b boot/grub/stage2_eltorito    \
                -no-emul-boot                   \
                -boot-load-size 4               \
                -A os                           \
                -input-charset utf8             \
                -quiet                          \
                -boot-info-table                \
                -o os.iso                       \
                iso
```

```
run: os.iso
    bochs -f bochsrc.txt -q

%.o: %.c
    $(CC) $(CFLAGS)  $< -o $@

%.o: %.s
    $(AS) $(ASFLAGS) $< -o $@

clean:
    rm -rf *.o kernel.elf os.iso
```

If your directory now looks like

```
.
|-- bochsrc.txt
|-- iso
|   |-- boot
|     |-- grub
|       |-- menu.lst
|       |-- stage2_eltorito
|-- kmain.c
|-- loader.s
```

then you should be able to start the OS in Bochs with the simple command

```
make run
```

Further Reading

- Kernigan & Richies book, *The C Programming Language, Second Edition*, [6] is great for learning about all the aspects of C

Chapter 4

Output

After booting our very small operating system kernel, the next task will be to be able to display text on the console. In this chapter we will create our first *drivers*, that is, code that acts as an layer between our kernel and the hardware, providing a nicer abstraction than speaking directly to the hardware. The first part of this chapter creates a driver for the framebuffer [23] to be able to display text to the user. The second part will create a driver for the serial port, since Bochs can store output from the serial port in a file, effectively creating a logging mechanism for our operating system!

Talking to the Hardware

There are usually two different ways to "talk" to the hardware, *memory-mapped I/O* and *I/O ports*.

If the hardware uses memory-mapped I/O, then you can write to a specific memory address and the hardware will be automatically get the new data. One example of this is the framebuffer, which will be discussed more in detail later. If you write the value `0x410F` to address `0x000B8000`, you will see the letter A in white color on a black background (see the section on the framebuffer for more details).

If the hardware uses I/O ports, then the assembly instructions `out` and `in` must be used to communicate with the device. `out` takes two parameters, the address of the I/O port and the value to send. `in` takes a single parameter, simply address of the I/O port and returns data. One can think of I/O ports as communicating with the hardware the same way as you communicate with a server using sockets. The cursor (the blinking rectangle) of the framebuffer is one example of hardware controlled via I/O ports.

The Framebuffer

The framebuffer is a hardware device that is capable of displaying a buffer of memory on the screen [23]. The framebuffer has 80 columns and 25 rows, and their indices start at 0 (so rows are labelled 0 - 24).

Writing Text

Writing text to the console via the framebuffer is done by memory-mapped I/O. The starting address of memory-mapped I/O for the framebuffer is `0x000B8000`. The memory is divided into 16 bit cells, where the 16 bits determines both the character, the foreground color and the background color. The highest eight

bits is the ASCII [24] value of the character, bit 7 - 3 the background and bit 3 - 0 the foreground, as can be seen in the following figure:

```
Bit:     | 15 14 13 12 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
Content: | ASCII                 | FG      | BG      |
```

The available colors are:

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

The first cell corresponds to row zero, column zero on the console. Using an ASCII table, one can see that A corresponds to 65 or `0x41`. Therefore, to write the character A with a green foreground (2) and dark grey background (8) at place (0,0), the following assembly instruction is used

```
mov [0x000B8000], 0x4128
```

The second cell then corresponds to row zero, column one and it's address is

```
0x000B8000 + 16 = 0x000B8010
```

This can all be done a lot easier in C by treating the address `0x000B8000` as a char pointer, `char *fb = (char *) 0x000B8000`. Then, writing A to at place (0,0) with green foreground and dark grey background becomes:

```
fb[0] = 'A';
fb[1] = 0x28;
```

This can of course be wrapped into a nice function

```
/** fb_write_cell:
 *  Writes a character with the given foreground and background to position i
 *  in the framebuffer.
 *
 *  @param i  The location in the framebuffer
 *  @param c  The character
 *  @param fg The foreground color
 *  @param bg The background color
 */
void fb_write_cell(unsigned int i, char c, unsigned char fg, unsigned char bg)
{
    fb[i] = c;
    fb[i + 1] = ((fg & 0x0F) << 4) | (bg & 0x0F)
}
```

which can then be used

```
#define FB_GREEN     2
#define FB_DARK_GREY 8

fb_write_cell(0, 'A', FB_GREEN, FB_DARK_GREY);
```

Moving the Cursor

Moving the cursor of the framebuffer is done via two different I/O ports. The cursor position is determined via a 16 bits integer, 0 means row zero, column zero, 1 means row zero, column one, 80 means row one, column zero and so on. Since the position is 16 bits large, and the `out` assembly instruction only take 8 bits as data, the position must be sent in two turns, first 8 bits then the next 8 bits. The framebuffer has two I/O ports, one for accepting the data, and one for describing the data being received. Port `0x3D4` is the command port that describes the data and port `0x3D5` is for the data itself.

To set the cursor at row one, column zero (position `80 = 0x0050`), one would use the following assembly instructions

```
out 0x3D4, 14      ; 14 tells the framebuffer to expect the highest 8 bits of the position
out 0x3D5, 0x00    ; sending the highest 8 bits of 0x0050
out 0x3D4, 15      ; 15 tells the framebuffer to expect the lowest 8 bits of the position
out 0x3D5, 0x50    ; sending the lowest 8 bits of 0x0050
```

The `out` assembly instruction can't be done in C, therefore it's a good idea to wrap it an assembly function which can be accessed from C via the cdecl calling standard [22]:

```
global outb             ; make the label outb visible outside this file

; outb - send a byte to an I/O port
; stack: [esp + 8] the data byte
;        [esp + 4] the I/O port
;        [esp    ] return address
outb:
    mov al, [esp + 8]    ; move the data to be sent into the al register
    mov dx, [esp + 4]    ; move the address of the I/O port into the dx register
    out dx, al           ; send the data to the I/O port
    ret                  ; return to the calling function
```

By storing this function in a file called `io.s` and also creating a header `io.h`, the `out` assembly instruction can now be conveniently accessed from C:

```
#ifndef INCLUDE_IO_H
#define INCLUDE_IO_H

/** outb:
 *  Sends the given data to the given I/O port. Defined in io.s
 *
 *  @param port The I/O port to send the data to
 *  @param data The data to send to the I/O port
```

```
 */
void outb(unsigned short port, unsigned char data);

#endif /* INCLUDE_IO_H */
```

Moving the cursor can now be wrapped in a C function:

```
#include "io.h"

/* The I/O ports */
#define FB_COMMAND_PORT         0x3D4
#define FB_DATA_PORT            0x3D5

/* The I/O port commands */
#define FB_HIGH_BYTE_COMMAND    14
#define FB_LOW_BYTE_COMMAND     15

/** fb_move_cursor:
 *  Moves the cursor of the framebuffer to the given position
 *
 *  @param pos The new position of the cursor
 */
void fb_move_cursor(unsigned short pos)
{
    outb(FB_COMMAND_PORT, FB_HIGH_BYTE_COMMAND);
    outb(FB_DATA_PORT,    ((pos >> 8) & 0x00FF));
    outb(FB_COMMAND_PORT, FB_LOW_BYTE_COMMAND);
    outb(FB_DATA_PORT,    pos & 0x00FF);
}
```

The Driver

Now that the basic functions, it's time to think about a driver interface for the framebuffer. There is no right or wrong about what functionality the interface should provide, but one suggestion is to have a `write` function with the declaration

```
int write(char *buf, unsigned int len);
```

that writes the contents of the buffer to the screen. The `write` function would automatically advance the cursor after a character has been written and also scroll the screen if necessary.

The Serial Ports

The serial port [25] is an interface for communicating between hardware devices, and it's usually not available on modern computers. However, the serial port is easy to use, and more importantly, can be used as a logging utility together with Bochs. A computer usually has several serial ports, but we will only make use of one of the ports, since we only will use it for logging. Furthermore, we will only use it for output, not input. The serial ports are controlled completely via I/O ports.

Configuring the Serial Port

The first data that needs to be sent to the serial port is some configuration data. In order for two hardware devices to be able to talk each other, they must agree upon some things. These things include:

- The speed used when sending data (bit or baud rate)
- If any error checking is used for the data (parity bit, stop bits)
- The number of bits that represents a unit of data (data bits)

Configuring the Line

Configuring the line means to configure how data is being sent over the line. The serial port has an I/O port, the *line command port* that is used for configuring the line.

First, the speed for sending data will be set. The serial port has an internal clock that runs at 115200 Hz. Setting the speed means sending a divisor to the serial port, for example sending 2 results in a speed of `115200 / 2 = 57600` Hz.

The divisor is a 16 bit number, but we can only send 8 bits at a time. Therefore, we must first send an instruction telling the serial port to expect first the highest 8 bits, then the lowest. This is done by sending `0x80` to the send line command port. The code then becomes:

```
#include "io.h" /* io.h is implement in the section "Moving the cursor" */

/* The I/O ports */

/* All the I/O ports are calculated relative to the data port. This is because
 * all serial ports (COM1, COM2, COM3, COM4) have their ports in the same
 * order, but they start at different values.
 */

#define SERIAL_COM1_BASE                0x3F8      /* COM1 base port */

#define SERIAL_DATA_PORT(base)          (base)
#define SERIAL_FIFO_COMMAND_PORT(base)  (base + 2)
#define SERIAL_LINE_COMMAND_PORT(base)  (base + 3)
#define SERIAL_MODEM_COMMAND_PORT(base) (base + 4)
#define SERIAL_LINE_STATUS_PORT(base)   (base + 5)

/* The I/O port commands */

/* SERIAL_LINE_ENABLE_DLAB:
 * Tells the serial port to expect first the highest 8 bits on the data port,
 * then the lowest 8 bits will follow
 */
#define SERIAL_LINE_ENABLE_DLAB         0x80

/** serial_configure_baud_rate:
 *  Sets the speed of the data being sent. The default speed of a serial
 *  port is 115200 bits/s. The argument is a divisor of that number, hence
```

```
 *  the resulting speed becomes (115200 / divisor) bits/s.
 *
 *  @param com      The COM port to configure
 *  @param divisor  The divisor
 */
void serial_configure_baud_rate(unsigned short com, unsigned short divisor)
{
    outb(SERIAL_LINE_COMMAND_PORT(com),
         SERIAL_LINE_ENABLE_DLAB);
    outb(SERIAL_DATA_PORT(com),
         (divisor >> 8) & 0x00FF);
    outb(SERIAL_DATA_PORT(com),
         divisor & 0x00FF);
}
```

Next, the way data is being sent must be configured, this is also done via the line command port by sending 8 bits. The layout of the 8 bits are as follows

```
Bit:     | 7 | 6 | 5 4 3 | 2 | 1 0 |
Content: | d | b | prty  | s | exp |
```

The content is

Name	Description
d	Enables (`d = 1`) or disables (`d = 0`) DLAB
b	If break control is enabled (`b = 1`) or disabled (`b = 0`)
prty	The number of parity bits to use (`prty = 0, 1, 2 or 3`)
s	The number of stop bits to use (`s = 0` equals 1, `s = 1` equals 1.5 or 2)
exp	Describes the length of the data, the length is `2^exp` (`exp = 3` results in 8 bits)

We will use the mostly standard value `0x03` [26], meaning a length of 8 bits, no parity bit, one stop bit and break control disabled. This is sent to the line command port, resulting in

```
/** serial_configure_line:
 *  Configures the line of the given serial port. The port is set to have a
 *  data length of 8 bits, no parity bits, one stop bit and break control
 *  disabled.
 *
 *  @param com  The serial port to configure
 */
void serial_configure_line(unsigned short com)
{
    /* Bit:     | 7 | 6 | 5 4 3 | 2 | 1 0 |
     * Content: | d | b | prty  | s | exp |
     * Value:   | 0 | 0 | 0 0 0 | 0 | 1 1 | = 0x03
     */
    outb(SERIAL_LINE_COMMAND_PORT(com), 0x03);
}
```

For a more in-depth explanation of the values, see [26].

Configuring the FIFO Queues

When data is transmitted via the serial port, it is placed in buffers, both when receiving and sending data. This way, if you send data to the serial port faster than it can send it over the wire, it will be buffered. However, if you send too much data too fast, the buffer will be full and then data will be dropped. The FIFO queue configuration byte looks like

```
Bit:     | 7 6 | 5  | 4 | 3   | 2   | 1   | 0 |
Content: | lvl | bs | r | dma | clt | clr | e |
```

The content is

Name	Description
lvl	How many bytes should be stored in the FIFO buffers
bs	If the buffers should 16 or 64 bytes large
r	Reserved
dma	How the serial port data should be accessed
clt	Clear the transmission FIFO buffer
clr	Clear the receiver FIFO buffer
e	If the FIFO buffer should be enabled or not

We use the value `0xC7 = 11000111` that:

```
- Enables FIFO
- Clear both receiver and transmission FIFO queues
- Use 14 bytes as size of queue
```

For a more in-depth explanation of the values, see [27]

Configuring the modem

The modem control register is used for very simple hardware flow control via the Ready To Transmit (RTS) and Data Terminal Ready (DTR) pins. When configuring the serial port before sending data, we want RTS and DTR to be 1.

The modem configuration byte looks like

```
Bit:     | 7 | 6 | 5  | 4  | 3   | 2   | 1   | 0   |
Content: | r | r | af | lb | ao2 | ao1 | rts | dtr |
```

The contents is

Name	Description
r	Reserved
af	Autoflow control enabled
lb	Loopback mode (used for debugging serial ports)
ao2	Auxiliary output 2, used for receiving interrupts
ao1	Auxiliary output 1
rts	Ready To Transmit
dtr	Data Terminal Ready

Since we don't need interrupts, because we won't handle any received data, we use the configuration value `0x03 = 00000011`, that is, RTS is set to 1 and DTR is set to 1.

Writing to the Serial Port

Writing to the serial port is done via the data I/O port. However, before writing, we must first ensure that the transmit FIFO queue is empty (so that all previous writes has been done). This is done by checking bit 5 of the line status I/O port.

Reading the contents of an I/O port is done via the `in` assembly instruction. There is no way to use the `in` instruction from C, so it has to be wrapped (the same way as the `out` instruction):

```
global inb

; inb - returns a byte from the given I/O port
; stack: [esp + 4] The address of the I/O port
;        [esp    ] The return address
inb:
    mov dx, [esp + 4]       ; move the address of the I/O port to the dx register
    in  al, dx              ; read a byte from the I/O port and store it in the al register
    ret                     ; return the read byte
```

```
/* in file io.h */

/** inb:
 *  Read a byte from an I/O port.
 *
 *  @param port The address of the I/O port
 *  @return     The read byte
 */
unsigned char inb(unsigned short port);
```

Checking if the transmit FIFO is empty can now be done from C:

```
#include "io.h"

/** serial_is_transmit_fifo_empty:
```

```
 *  Checks whether the transmit FIFO queue is empty or not for the given COM
 *  port.
 *
 *  @param com The COM port
 *  @return 0 if the transmit FIFO queue is not empty
 *          1 if the transmit FIFO is empty
 */
int serial_is_transmit_fifo_empty(unsigned int com)
{
    /* 0x20 = 0001 0000 */
    return inb(SERIAL_LINE_STATUS_PORT(com)) & 0x20;
}
```

Now, the writing to serial port means spinning while the transmit FIFO queue isn't empty, and then writing to the data port.

Configuring Bochs

To use the serial port together with Bochs, you must tell Bochs via the `bochsrc.txt` file to save the output from the serial port to a file. This can be done with the `com1` config parameter:

```
com1: enabled=1, mode=file, dev=com1.out"
```

The output will now be stored in the `com1.out` file.

The Driver

We recommend that you try to write a `printf` like function, see section 7.3 in [6]. We also recommend that you create some way of distinguish the severeness of log message, for example by prepending the messages with `DEBUG`, `INFO` or `ERROR`.

Further Reading

- The book "Serial programming" (available on WikiBooks) has a great section on programming the serial port, `http://en.wikibooks.org/wiki/Serial_Programming/8250_UART_Programming#UART_Registers`
- The OSDev wiki has a page with a lot of information about the serial ports, `http://wiki.osdev.org/Serial_ports`

Chapter 5

Segmentation

Segmentation in x86 means accessing the memory through segments. Segments are portions of the address space, possibly overlapping, specified by a base address and a limit. To address a byte in segmented memory, you use a 48-bit *logical address*: 16 bits that specifies the segment, and 32-bits to specify what offset within that segment you want. The offset is added to the base address of the segment, and the resulting linear address is checked against the segment's limit - see the figure below. If everything works out fine, (including access-right checks ignored for now), the results is a *linear address*. If paging is disabled (see the chapter on paging, this linear address space is mapped 1:1 on the *physical address* space, and the physical memory can be accessed.

To enable segmentation you need to set up a table that describes each segment - a segment descriptor table. In x86, there are two types of descriptor tables: The global descriptor table (GDT) and local descriptor tables (LDT). An LDT is set up and managed by user-space processes, and each process have their own LDT. LDT's can be used if a more complex segmentation model is desired - we won't use it. The GDT shared by everyone - it's global.

Accessing Memory

Most of the time when we access memory we don't have to explicitly specify the segment we want to use. The processor has six 16-bit segment registers: `cs`, `ss`, `ds`, `es`, `gs` and `fs`. `cs` is the code segment register and specifies the segment to use when fetching instructions. `ss` is used whenever the accessing the stack (through the stack pointer `esp`), and `ds` is used for other data accesses. `es`, `gs` and `fs` are free to use however we or the user processes wish (we will set them, but not use them).

Implicit use of segment registers:

```
func:
    mov eax, [esp+4]
    mov ebx, [eax]
    add ebx, 8
    mov [eax], ebx
    ret
```

Explicit version:

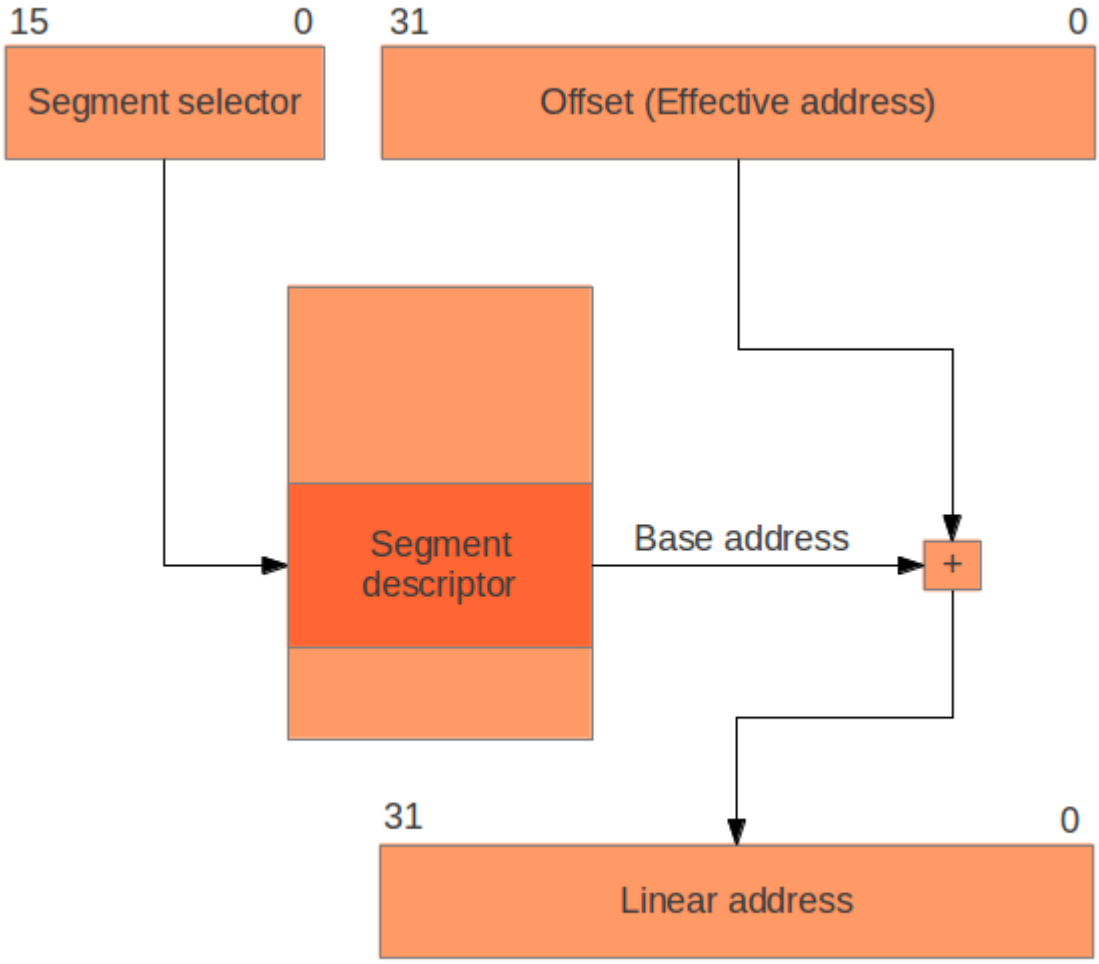

Figure 5.1: Translation of logical addresses to linear addresses.

```
func:
    mov eax, [ss:esp+4]
    mov ebx, [ds:eax]
    add ebx, 8
    mov [ds:eax], ebx
    ret
```

(You don't need to use `ss` for storing the stack segment selector, or `ds` for the data segment selector, but it is convenient, and makes it possible to use the implicit style above.)

Segment descriptors and their fields are described in figure 3-8 in the Intel manual [28].

The Global Descriptor Table (GDT)

A GDT/LDT is an array of 8-byte segment descriptors. The first descriptor in the GDT is always a null descriptor, and can never be used to access memory. We need at least two segment descriptors (plus the null descriptor) for our GDT (and two more later when we enter user mode, see the segments for user mode section). This is because the descriptor contains more information than just the base and limit fields. The two most relevant fields here are the *Type* field and the *Descriptor Privilege Level* (DPL) field.

The table 3-1, chapter 3, in the Intel manual [28] specifies the values for the Type field, and it is because of this that we need at least two segments: One segment to execute code (to put in `cs`) (Execute-only or Execute-Read) and one to read and write data (Read/Write) (to put in the other segment registers).

The DPL specifies the privilege levels required to execute in this segment. x86 allows for four privilege levels (PL), 0 to 3, where PL0 is the most privileged. In most operating systems (eg. Linux and Windows), only PL0 and PL3 are used (although MINIX uses all levels). The kernel should be able to do anything, so it "runs in PL0" (uses segments with DPL set to 0), and user-mode processes should run in PL3. The current privilege level (CPL) is determined by the segment selector in `cs`.

Since we are now executing in kernel mode (PL0), the DPL should be 0. The segments we need are described in the table below.

Index	Offset	Name	Address range	Type	DPL
0	`0x00`	null descriptor			
1	`0x08`	kernel code segment	`0x00000000 - 0xFFFFFFFF`	RX	PL0
2	`0x10`	kernel data segment	`0x00000000 - 0xFFFFFFFF`	RW	PL0

Table 5.1: The segment descriptors needed.

Note that the segments overlap - they both encompass the entire linear address space. In our minimal setup we'll only use segmentation to get privilege levels. See the [28], chapter 3, for details on the other descriptor fields.

Creating and Loading the GDT

Creating the GDT can be done both in C and assembly. A static fixed-size array should do the trick.

To load the GDT into the processor we use the `lgdt` instruction, which takes the address of a struct that specifies the start and size of the GDT:

```
32-bit: start of GDT
16-bit: size of GDT (8 bytes * num entries)
```

It is easiest to encode this information using a "packed struct".

If `eax` has an address to such a struct, we can just to the following:

```
lgdt [eax]
```

It might be easier if you make this instruction available from C, the same way as was done with `in` and `out`.

Now that the processor knows where to look for the segment descriptors we need to load the segment registers with the corresponding segment selectors. The content of a segment selector is described in the table below.

```
Bit:     | 15                              3 | 2  | 1 0 |
Content: | offset (index)                     | ti | rpl |
```

Name	Description
rpl	Requested Privilege Level - we want to execute in PL0 for now.
ti	Table Indicator. 0 means that this specifies a GDT segment, 1 means an LDT Segment.
offset (index)	Offset within descriptor table.

Table 5.2: The layout of segment selectors.

The offset is added to the start of the GDT to get the address of the segment descriptor: `0x08` for the first descriptor and `0x10` for the second, since each descriptor is 8 bytes. The Requested Privilege Level (RPL) should be `0`, since we want to remain in PL0.

Loading the segment selector registers is easy for the data registers - just copy the correct offsets into the registers:

```
mov ds, 0x10
mov ss, 0x10
mov es, 0x10
; ...
```

To load `cs` we have to do a "far jump":

```
    ; using previous cs
    jmp 0x08:flush_cs

flush_cs:
    ; now we've changed cs to 0x08
```

A far jump is a jump where we explicitly specify the full 48-bit logical address: The segment selector to use, and the absolute address to jump to. It will first set `cs` to `0x08`, and then jump to `.flush_cs` using its absolute address.

Whenever we load a new segment selector into a segment register, the processor reads the entire descriptor and stores it in shadow registers within the processor.

Further Reading

- Chapter 3 of the Intel manual [28] is quite good; low-level and technical.
- The OSDev wiki can be useful: `http://wiki.osdev.org/Segmentation`
- The Wikipedia page on x86 segmentation might be worth looking into: `http://en.wikipedia.org/wiki/X86_memory_segmentation`

Chapter 6

Interrupts and Input

Now that the OS can produce *output*, it would be nice if it also could get some *input*. In order to read information from the keyboard, the operating system must be able to handle *interrupts*. An interrupt occurs when a hardware device, such as the keyboard, the serial port or the timer, signals the CPU that the state of the device has changed. The CPU itself can also send interrupts due to program errors, for example when a program references memory it doesn't has access to, or when a program divides a number by zero. Finally, there are also *software intterupts*, which are interrupts that are caused by the `int` assembly instruction, and they are often used for system calls (syscalls).

Interrupts Handlers

Interrupts are handled via the *interrupt descriptor table* (IDT). The IDT that describes a handler for each interrupt. Each interrupt has a number (0 - 255), and the handler for interrupt i is defined at the *i:th* position in the table. There are three different kinds of handlers for interrupts:

- Task handler
- Interrupt handler
- Trap handler

The task handler uses Intel specific functionality, so they won't be covered here (see the Intel manual [28], chapter 6, for more info). The only difference between an interrupt handler and a trap handler is that the interrupt handler disables interrupts, that is, you can't get an interrupt while at the same time handling an interrupt. In this book, we will use trap handlers and disables interrupts manually when we need to.

Creating an Entry in the IDT

An entry in the IDT for an interrupt handler consists of 64 bits. The highest 32 bits are

```
Bit:     | 31              16 | 15 | 14 13 | 12 | 11 | 10 9 8 | 7 6 5 | 4 3 2 1 0 |
Content: | offset high        | P  | DPL   | 0  | D  | 1  1 0 | 0 0 0 | reserved  |
```

and the lowest 32 bits are

```
Bit:     | 31              16 | 15              0 |
Content: | segment selector   | offset low        |
```

where the content is

Name	Description
offset high	The 16 highest bits of the 32 bit address in the segment
offset low	The 16 lowest bits of the 32 bits address in the segment
p	If the handler is present in memory or not (1 = present, 0 = not present)
DPL	Descriptor Privilige Level, the privilege level the handler can be called from (0, 1, 2, 3)
D	Size of gate, (1 = 32 bits, 0 = 16 bits)
segment selector	The index in the GDT
r	Reserved

The offset is a pointer to code (preferably an assembly label). For example, to create an entry for a handler which code starts at `0xDEADBEEF` that runs in privilege level 0 (same as the kernel) and therefore uses the same code segment selector as the kernel, the following two bytes would be used:

```
0xDEAD8E00
0x0008BEEF
```

If the IDT then is represented an `unsigned integer idt[256];` then to register this as the handler for interrupt 0 (divide error), you would write

```
idt[0] = 0xDEAD8E00
idt[1] = 0x0008BEEF
```

As written in the chapter getting to C, we recommend that you instead of using bytes (or unsigned integers) use packed structures to make the code more readable.

Creating a Generic Interrupt Handler

When the interrupt occurs, the CPU will push information about the interrupt on the stack, then look up the appropriate interrupt hander in the IDT and jump to it. Once the interrupt handler is done, it uses the `iret` instruction to return. `iret` pops the information from the stack, restores EFLAGS and jumps to `CS:EIP`.

The stack at the time of the interrupt will look like

```
[esp + 12] EFLAGS
[esp + 8]  CS
[esp + 4]  EIP
[esp]      Error code?
```

The reason for the question mark behind error code is because not all interrupts creates an error code. The CPU interrupts that put an error code on the stack are 8, 10, 11, 12, 13, 14 and 17.

The interrupt handler has to be written in assembly, because the all the registers has to be pushed on the stack, because the code that got interrupted will assume that the state of its registers hasn't changed. Writing all the logic of the interrupt handler in assembly will be tiresome, so creating an assembly handler that saves the registers, calls a C function, restores the registers and returns is a good idea!

The C handler should get the state of the registers, the state of the stack and the number of the interrupt. The following definitions can for example be used:

```
struct cpu_state {
    unsigned int eax;
    unsigned int ebx;
    unsigned int ecx;
    .
    .
    .
    unsigned int esp;
} __attribute__((packed));

struct stack_state {
    unsigned int error_code;
    unsigned int eip;
    unsigned int cs;
    unsigned int eflags;
} __attribute__((packed));

void interrupt_handler(struct cpu_state cpu, unsigned int interrupt, struct stack_state stack);
```

Unfortunately, the CPU does *not* push the interrupt number on the stack. Since writing one version for each interrupt is tedious, it's better to use the macro functionality of NASM (see [29] for more details). Since not all interrupts produce an error code, the error code 0 will be used for interrupts without error code.

```
%macro no_error_code_interrupt_handler %1
global interrupt_handler_%1
interrupt_handler_%1:
    push    dword 0                     ; push 0 as error code
    push    dword %1                    ; push the interrupt number
    jmp     common_interrupt_handler    ; jump to the common handler
%endmacro

%macro error_code_interrupt_handler %1
global interrupt_handler_%1
interrupt_handler_%1:
    push    dword %1                    ; push the interrupt number
    jmp     common_interrupt_handler    ; jump to the common handler
%endmacro

common_interrupt_handler:               ; the common parts of the generic interrupt handler
    ; save the registers
    push    eax
```

```
    push    ebx
    .
    .
    .
    push    ebp

    ; call the C function
    call    interrupt_handler

    ; restore the registers
    pop     ebp
    .
    .
    .
    pop     ebx
    pop     eax

    ; restore the esp
    add     esp, 8

    ; return to the code that got interrupted
    iret

no_error_code_interrupt_handler 0       ; create handler for interrupt 0
no_error_code_interrupt_handler 1       ; create handler for interrupt 1
.
.
.
error_code_handler              7       ; create handler for interrupt 7
.
.
.
```

The `common_interrupt_handler` does the following: - Push the registers onto the stack - Call the C function `interrupt_handler` - Pop the registers from the stack - Add 8 to `esp` (due to the error code and the interrupt number) - Execute `iret` to return to the interrupted code

Since the macros declares global labels, the addresses of the interrupt handlers can be accessed from C or assembly when creating the IDT.

Loading the IDT

The IDT is loaded with `lidt` assembly instruction which takes the address of the first element in the array. It is easiest to wrap this instruction and instead use it from C:

```
global  load_idt

; load_idt - Loads the interrupt descriptor table (IDT).
; stack: [esp + 4] the address of the first entry in the IDT
;        [esp    ] the return address
```

```
load_idt:
    mov     eax, [esp+4]    ; load the address of the IDT into register eax
    lidt    eax             ; load the IDT
    ret                     ; return to the calling function
```

Programmable Interrupt Controller (PIC)

To start using hardware interrupts, you must first configure the Programmable Interrupt Controller (PIC). The PIC makes it possible to map signals from the hardware to interrupts. The reasons for configuring the PIC are:

- Remap the interrupts. By default, the PIC uses interrupts 0 - 15 for hardware interrupts, which conflicts with the CPU interrupts, so the PIC interrupts must be remapped to another interval.
- Select which interrupts to receive. You probably don't want to receive interrupts from all hardware, since you don't have code that manages these interrupts anyway.
- Set up the correct mode for the PIC.

In the beginning, there was only one PIC (PIC 1) and eight interrupts. As more hardware got added in the computer, 8 interrupts were too few. The solution was then to chain on another PIC (PIC 2) on the first PIC (see interrupt 2 on PIC 1).

The hardware interrupts are:

PIC 1	Hardware	PIC 2	Hardware
0	Timer	8	Real Time Clock
1	Keyboard	9	General I/O
2	PIC 2	10	General I/O
3	COM 2	11	General I/O
4	COM 1	12	General I/O
5	LPT 2	13	Coprocessor
6	Floppy disk	14	IDE Bus
7	LPT 1	15	IDE Bus

A great tutorial for configuring the PIC can be found at [30].

Every interrupt from the PIC has to be acknowledged, that is, sending a message to the PIC confirming that the interrupt has been handled. If this isn't done, the PIC won't generate any more interrupts.

This is done by sending the byte `0x20` to the PIC that raised the interrupt. Sending the acknowledgement to the PIC that did *not* raise the interrupt doesn't do anything. Therefore, implementing a `pic_acknowledge` function is straightforward:

```
#include "io.h"
```

```
#define PIC1_PORT_A 0x20
#define PIC2_PORT_A 0xA0

#define PIC_ACK     0x20

/** pic_acknowledge:
 *  Acknowledges an interrupt from either PIC 1 or PIC 2.
 */
void pic_acknowledge(void)
{
    outb(PIC1_PORT_A, PIC_ACK);
    outb(PIC2_PORT_A, PIC_ACK);
}
```

Reading Input from the Keyboard

The keyboard generates interrupt 1 from the PIC, so if you remapped the PIC's interrupts to start at number 32, the interrupt number for the keyboard will be 33.

The keyboard does not generate ASCII characters, instead it generates scan codes. A scan code represents a button. The scan code representing the just pressed button can be read from the keyboard's data I/O port:

```
#include "io.h"

#define KBD_DATA_PORT   0x60

/** read_scan_code:
 *  Reads a scan code from the keyboard
 *
 *  @return The scan code (NOT an ASCII character!)
 */
unsigned char read_scan_code(void)
{
    return inb(KBD_DATA_PORT);
}
```

The next step is to write a function that translates a scan code to the corresponding ASCII character. If you want to map the scan codes to the ASCII characters as is done on an American keyboard, [31] shows how the scan codes and keyboard buttons correlates.

Remember, since the keyboard interrupt is raised by the PIC, you must call `pic_acknowledge` at the end of the keyboard interrupt handler!

Further Reading

- The OSDev wiki has a great page on interrupts, `http://wiki.osdev.org/Interrupts`
- Chapter 6 of Intel Manual 3a [28] describes everything there is to know about interrupts.

Chapter 7

The Road to User Mode

Now that the kernel boots, prints to screen and reads from keyboard - what do we do? Usually, a kernel is not supposed to do the application logic itself, but leave that for applications. The kernel creates the proper abstractions (for memory, files, devices, etc.) so that application development becomes easier, performs tasks on behalf of applications (system calls), and schedules processes.

User mode, in contrast with kernel mode, is the environment in which the user's programs execute. It is less privileged than the kernel, and will prevent badly written user programs from messing with other programs or the kernel. Badly written kernels are free to mess up what they want.

There's quite a way to go until we get there, but here follows a quick-and-dirty start.

Loading a Program

Where do we get the application from? Somehow we need to load the code we want to execute into memory. More feature complete operating system usually have drivers and file system that enables them load the software from a CD-ROM drive, a hard disk or a floppy disk.

Instead of creating all these drivers and file systems, we will use a feature in GRUB called modules to load our program.

GRUB Modules

GRUB can load arbitrary files into memory from the ISO image and these files are usually referred to as modules. To make GRUB load a module, edit the file `iso/boot/grub/menu.lst` and add the following line at the end of the file

```
module /modules/program
```

Now create the folder `iso/modules`

```
mkdir -p iso/modules
```

Later in this chapter, the application "program" will be created. We must also update the code that calls `kmain` to pass information to `kmain` about where it can find the modules. We also want to tell GRUB that

it should align all the modules on page boundaries when loading them (see the chapter about "Paging" for details about page alignment).

To instruct GRUB how to load our modules, the "multiboot header", that is the first bytes of the kernel, must be updated as follows:

```
; in file `loader.s`


MAGIC_NUMBER    equ 0x1BADB002      ; define the magic number constant
ALIGN_MODULES   equ 0x00000001      ; tell GRUB to align modules

; calculate the checksum (all options + checksum should equal 0)
CHECKSUM        equ -(MAGIC_NUMBER + ALIGN_MODULES)

section .text:                      ; start of the text (code) section
align 4                             ; the code must be 4 byte aligned
    dd MAGIC_NUMBER                 ; write the magic number
    dd ALIGN_MODULES                ; write the align modules instruction
    dd CHECKSUM                     ; write the checksum
```

GRUB will also store a pointer to a `struct` in the register `ebx` that among other things describes at which addresses the modules are loaded. Therefore, you probably want to push `ebx` on the stack before calling `kmain` to make into an argument for `kmain`.

Executing a Program

A Very Simple Program

Any program we write at this stage won't be able to communicate with the outside. Therefore, a very short program that writes a value to a register suffices. Halting Bochs and reading its log should verify that the program has run.

```
; set eax to some distinguishable number, to read from the log afterwards
mov eax, 0xDEADBEEF

; enter infinite loop, nothing more to do
; $ means "beginning of line", ie. the same instruction
jmp $
```

Compiling

Since our kernel cannot parse advanced executable formats, we need to compile the code into a flat binary. NASM can do it with the flag `-f`:

```
nasm -f bin program.s -o program
```

This is all we need. You must now move the file `program` to the folder `iso/modules`.

Finding the Program in Memory

Before jumping to the program, we must find where it resides in memory. Assuming that the contents of the `ebx` is passed as an argument to `kmain`, we can do this entirely from C.

The pointer in `ebx` points to a multiboot-structure [17]. Download the `multiboot.h` file from `http://www.gnu.org/software/grub/manual/multiboot/html_node/multiboot.h.html`, which describes the structure.

The pointer passed to `kmain` from the `ebx` register can now be casted to `multiboot_info_t` pointer. The address of the first module is now in the field `mods_addr`, that is

```
int kmain(..., unsigned int ebx)
{
    multiboot_info_t *mbinfo = (multiboot_info_t *) ebx;
    unsigned int address_of_module = mbinfo->mods_addr;
}
```

However, before just blindly following the pointer, you should check that the module got loaded correctly. This can be done by checking the `flags` field of the `multiboot_info_t` stucture. You should also check the field `mods_count` to make sure it's exactly 1. For more details about the multiboot structure, see [17].

Jumping to the Code

What we'd like to do now is just jump to the address of the GRUB-loaded module. Since it is easier to parse the multiboot structure in C, calling the code from C is more convenient, but it can of course be done with `jmp` (or `call`) in assembly as well.

```
typedef void (*call_module_t)(void);
/* ... */
call_module_t start_program = (call_module_t) address_of_module;
start_program();
/* we'll never get here, unless the module code returns */
```

If we start the kernel, wait until it has run and entered the infinite loop in the program, and then halt Bochs, we should see `0xDEADBEEF` in `eax` via the Bochs log. We have successfully started a program in our OS!

The Beginning of User Mode

The program we've written now runs in the same mode as the kernel, we've just entered it in a somewhat peculiar way. To enable applications to execute at a different privilege level, we'll need to do segmentation, paging and allocate page frames.

It's quite a lot of work and technical details to go through, but in a few chapters you'll have a working user mode programs.

Chapter 8

Virtual Memory, an Introduction

Virtual memory is an abstraction of physical memory. The purpose of virtual memory is generally to simplify application development and to let processes address more memory than what is actually physically present. We don't want applications messing with the kernel or other applications' memory.

In the x86 architecture, virtual memory can be accomplished in two ways: Segmentation and paging. Paging is by far the most common and versatile technique, and we'll implement it in chapter 7. Some use of segmentation is still necessary (to allow for code to execute under different privilege levels), so we'll set up a minimal segmentation structure in the next chapter.

Managing memory is a big part of what an operating system does. Paging and page frame allocation deals with that.

Segmentation and paging is described in the Intel manual [28], chapter 3 and 4.

Virtual Memory Through Segmentation?

You could skip paging entirely and just use segmentation for virtual memory. Each user mode process would get its own segment, with base address and limit properly set up so that no process can see someone else's memory. A problem with this is that all memory for a process needs to be contiguous. Either we need to know in advance how much memory the program will require (unlikely), or we can move the memory segments to places where they can grow when the limit is reached (expensive, causes fragmentation - can result in "out of memory" even though enough memory is available, but in too small chunks). Paging solves both these problems.

It might be interesting to note that in x86_64 (another CPU architecture), segmentation is almost completely removed.

Further Reading

- LWN.net has an article on virtual memory: `http://lwn.net/Articles/253361/`
- So does Gustavo Duarte: `http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation`

Chapter 9

Paging

Segmentation translates a logical address into a linear address. Paging translates these linear addresses onto the physical address space, and determines access rights and how the memory should be cached.

Why Paging?

Paging is the most common technique used in x86 to enable virtual memory. Virtual memory means that each process will get the impression that the available memory range is `0x00000000 - 0xFFFFFFFF` even though the actual size of the memory is way less. It also means that when a process addresses a byte of memory, they will use a virtual (linear) address instead of physical one. The code in the user process won't notice any difference (except for execution delays...). The linear address will then get translated to a physical address by the MMU and the page table. If the virtual address isn't mapped to a physical address, the CPU will raise a page fault interrupt.

Paging is optional, and some operating systems don't need it. But if we want memory access control (so that we can have processes running in different privilege levels), paging is the neatest way to do it.

Paging in x86

x86 paging (chapter 4 in the Intel manual [28]) consists of a page directory (PDT) that can contain references to 1024 page tables (PT), each of which can point to 1024 sections of physical memory called page frames (PF); Each page frame is 4096 byte large. In a virtual address (linear address), the highest 10 bits specifies the offset of a page directory entry (PDE) in the current PDT, the next highest 10 bits the offset of a page table entry (PTE) within the page table pointed to by that PDE. The lowest 12 bits in the address is the offset within the page frame to be addressed.

All page directories, page tables and page frames need to be aligned on 4096 byte addresses. This makes it possible to address a PDT, PT or PF with just the highest 20 bits of a 32 bit address, since the lowest 12 need to be zero.

The PDE and PTE structure is very similar to each other: 32 bits (4 bytes), where the highest 20 bits points to a PTE or PF, and the lowest 12 bits control access rights and other configurations. 4 bytes * 1024 equals 4096 bytes, so a page directory and page table both fit in a page frame themselves.

The translation of linear addresses to physical addresses is described in the figure below.

It is also possible to use 4MB pages. A PDE then points directly to a 4MB page frame, which needs to be aligned on a 4MB address boundary. The address translation is almost the same as in the figure, with just the page table step removed. It is possible to mix 4MB and 4KB pages.

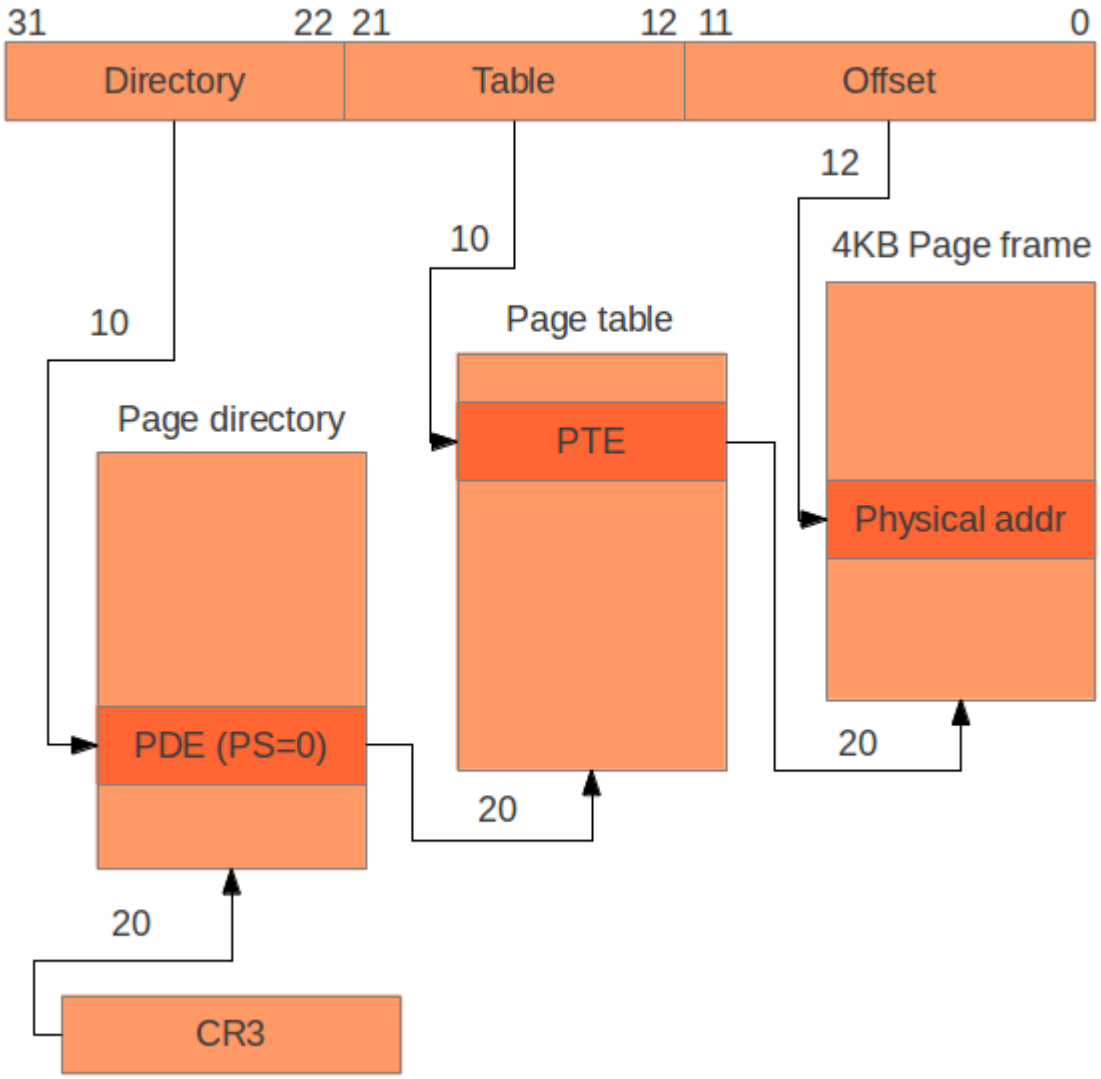

Figure 9.1: Translating virtual addresses (linear addresses) to physical addresses.

The 20 bits pointing to the current PDT is stored in the `cr3` register. The lower 12 bits of `cr3` are used configuration.

For more details on the paging structures, see chapter 4 in the Intel manual [28]. The most interesting bits are U/S, which determine what privilege levels can access this page (PL0 or PL3), and R/W, which makes the memory in the page read-write or read-only.

Identity Paging

The simplest kind of paging is when we map each virtual address onto the same physical address. This can be done at compile time by, for instance, creating a page table where each entry points to its corresponding 4MB frame. In NASM we can do this with macros and commands (`%rep`, `times` and `dd`), or just ordinary assembler instructions.

Enabling Paging

You enable paging by first writing the address of a page directory to `cr3` and then setting bit 31 (the PG "paging-enable" bit) of `cr0` to `1`. If you want 4MB pages, set the PSE bit (Page Size Extensions, bit 4) of `cr4`.

```
; eax has the address of the page directory
mov cr3, eax

mov ebx, cr4        ; read current cr4
or  ebx, 0x00000010 ; set PSE
mov cr4, ebx        ; update cr4

mov ebx, cr0        ; read current cr0
or  ebx, 0x80000000 ; set PG
mov cr0, ebx        ; update cr0

; now paging is enabled
```

A Few Details

It is important to note that all addresses within the page directory, page tables and in `cr3` needs to be physical addresses to the structures, never virtual. This will be more relevant in later sections where we dynamically update the paging structures, such as in the chapter on user mode.

An instruction that is useful when an updating a PDT or PT is `invlpg`. It invalidates the TLB (Translation Lookaside Buffer) for a virtual address. This is only required when changing a PDE or PTE that was previously mapped to something else. If it had previously been marked as not present (bit 0 was set to 0), doing `invlpg` is unnecessary. Also, changing the value of `cr3` will cause all entries in the TLB to be invalidated.

Example:

```
; invalidate any TLB references to virtual address 0
invlpg [0]
```

Higher-half Kernel

When GRUB loads our kernel into memory, it places the kernel at the physical memory starting at 1MB. With identity mapping, this is also the virtual address of the kernel.

Reasons to Not Identity Map the Kernel

If the kernel is placed at the beginning of the virtual address space - that is, the virtual address `0x00000000` to `"size of kernel"` will map to the location of the kernel in memory - there will be issues when linking the user mode process code. Normally, during linking, the linker assumes that the code will be loaded into the memory position `0x00000000`. Therefore, when resolving absolute references, `0x00000000` will be the base address for calculating the exact position. But if the kernel is mapped onto the virtual address space (`0x00000000`, `"size of kernel"`), the user mode process cannot be loaded at virtual address `0x00000000` - it must be placed somewhere else. Therefore, the assumption from the linker that the user mode process is loaded into memory at position `0x00000000` is wrong. This can be corrected by using a linker script which tells the linker to assume a different starting address, but that is a very cumbersome solution for the users of the operating system.

This also assumes that we want the kernel to be part of the user mode process' address space. As we will see later, this is a nice feature, since during system calls we don't have to change any paging structures to get access to the kernels code and data. The kernel pages will of course require privilege level 0 for access, to prevent a user process from reading or writing kernel memory.

The Virtual Address for the Kernel

Preferably, the kernel should be placed at a very high virtual memory address, for example `0xC0000000` (3 GB). The user mode process is not likely to be 3GB large, which is now the only way that it can conflict with the kernel.

If the user mode process is larger than 3GB, some pages will need to be swapped out by the kernel. Swapping pages will not be part of this book.

Placing the Kernel at `0xC0100000`

Placing the kernel at `0xC0100000` isn't hard, but it does require some thought. This is once again a linking problem. When the linker resolves all absolute references in the kernel, it will assume that our kernel is loaded at physical memory location `0x00100000`, not `0x00000000`, since we tell it so in our linker script (see the section on linking the kernel. However, we want the jumps to be resolved to `0xC0100000` as base, since otherwise a kernel jump will jump straight into the user mode process code (remember that the user mode process is loaded at virtual memory `0x00000000` and up).

However, we can't simply tell the linker to assume that the kernel starts (is loaded) at `0xC01000000`, since we want it to be loaded at the physical address `0x00100000`. The reason for having the kernel loaded at 1 MB is because it can't be loaded at `0x00000000`, since there is BIOS and GRUB code loaded below 1 MB. We also cannot assume that we can load the kernel at `0xC0100000`, since the machine might not have 3 GB of physical memory.

This can be solved by using both relocation (`.=0xC0100000`) and the `AT` instruction in the linker script. Relocation specifies that non-relative memory-references should should use the relocation address as base in address calculations. `AT` specifies where the kernel should be loaded into memory. Relocation is done at link time by `ld` [32], the load address specified by `AT` is handled by GRUB when loading the kernel, and is part of the ELF format [16].

Higher-half Linker Script

We can modify the first linker script to implement this:

```
ENTRY(loader)           /* the name of the entry symbol */

. = 0xC0100000          /* the code should be relocated to 3GB + 1MB */

/* align at 4KB and load at 1MB */
.text ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(.text)            /* all text sections from all files */
}

/* align at 4KB and load at 1MB + . */
.rodata ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(.rodata*)         /* all read-only data sections from all files */
}

/* align at 4KB and load at 1MB + . */
.data ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(.data)            /* all data sections from all files */
}

/* align at 4KB and load at 1MB + . */
.bss ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(COMMON)           /* all COMMON sections from all files */
    *(.bss)             /* all bss sections from all files */
}
```

Entering the Higher Half

When GRUB jumps to the kernel code, there is no paging table. Therefore, all references to `0xC0100000 + X` won't be mapped to the correct physical address, and will therefore cause a general protection exception (GPE) at the very best, otherwise (if the user has more than 3 GB of memory) the OS will just crash.

Therefore, assembly code that doesn't use absolute jumps must be used to set up the page table, add one entry for the first 4 MB of the virtual address space that maps to the first 4 MB of the physical memory and finally an entry for `0xC0100000` that maps to `0x0010000`. If we skip the identity mapping for the first 4 MB, the CPU would generate a page fault immediately after paging was enabled when trying to fetch the next instruction from memory. After the table has been created, an indirect jump can be done to a label, to make `eip` point to a high address:

```
; assembly code executing at around 0x00100000
; enable paging for both actual location of kernel
; and its higher-half virtual location

lea ebx, [higher_half] ; load the address of the label in ebx
jmp ebx                ; jump to the label

higher_half:
    ; code here executes in the higher half kernel
```

```
; eip is larger than 0xC0000000
; can continue kernel initialisation, calling C code, etc.
```

Now `eip` will point to a memory location somewhere right after `0xC0100000` - all the code can now execute as if it were at `0xC0100000`, the higher-half. The entry mapping the first 4MB of virtual memory to the first 4MB of physical can be removed from the page table, and its corresponding entry in the TLB invalidated with `invlpg [0]`.

Running in the Higher Half

There are a few details we must deal with when using a higher-half kernel. We must now take care when using memory mapped I/O that uses specific memory locations. For example, the frame buffer is located at `0x000B8000`, but since there is no entry in the page table for the address `0x000B8000` any longer, the address `0xC00B8000` must be used, since the virtual address `0xC0000000` maps to the physical address `0x00000000`.

Any explicit references to addresses within the multiboot structure needs to be changed to reflect the new virtual addresses as well.

Mapping entire 4 MB pages for the kernel is very simple, but wastes quite some memory (unless you have a really big kernel). Creating a higher-half kernel mapped in as 4KB pages is doable, but somewhat trickier. The page directory and one page table can be in `.data` (assuming your kernel is smaller than 4 MB), but you need to configure the mappings at runtime. The size of the kernel can be determined by exporting labels from the linker script [32], which we'll need to do later anyway when writing the page frame allocator.

Virtual Memory Through Paging

Paging enables two things that are good for virtual memory. First, it allows for fine-grained access control to memory. You can mark pages as read-only, read-write, only for PL0 etc. Second, it creates the illusion of contiguous memory. User mode processes, and the kernel, can access memory as if it were contiguous, and the contiguous memory can be extended without the need to move stuff around. Also, we can allow the user mode programs to access all memory below 3GB, but unless they actually use it, we don't have to assign page frames to the pages. This allows processes to have code located near `0x00000000` and the stack at just below `0xC0000000`, and still not require more than two actual pages.

Further Reading

- Chapter 4 (and to some extent chapter 3) of the Intel manual [28] are your definitive sources for the details about paging.
- Wikipedia has an article on paging: `http://en.wikipedia.org/wiki/Paging`
- The OSDev wiki has a page on paging: `http://wiki.osdev.org/Paging` and a tutorial for making a higher-half kernel: `http://wiki.osdev.org/Higher_Half_bare_bones`
- Gustavo Duarte's article on how a kernel manages memory is well worth a read: `http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory`
- Details on the linker command language can be found in [32].
- More details on the ELF format can be found in this pdf: `http://flint.cs.yale.edu/cs422/doc/ELF_Format.pdf`

Chapter 10

Page Frame Allocation

Now that we have virtual memory with paging set up, we need to somehow allocate some of the available memory to use for the applications we want to run. How do we know what memory is free? That is the role of a page frame allocator.

Managing Available Memory

How Much Memory is There?

First we need to know how much memory the computer we're running on has. The easiest way to do this is to read it from the multiboot [17] structure sent to us by GRUB. GRUB collects the information we need about the memory - what is reserved, I/O mapped, read-only etc. We must make sure that we don't mark the part of memory used by the kernel as free. One way to do this is to export labels at the beginning and end of the kernel binary from the linker script.

The updated linker script:

```
ENTRY(loader)           /* the name of the entry symbol */

. = 0xC0100000          /* the code should be relocated to 3 GB + 1 MB */

/* these labels get exported to the code files */
kernel_virtual_start = .;
kernel_physical_start = . - 0xC0000000;

/* align at 4 KB and load at 1 MB */
.text ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(.text)            /* all text sections from all files */
}

/* align at 4 KB and load at 1 MB + . */
.rodata ALIGN (0x1000) : AT(ADDR(.rodata)-0xC0000000)
{
    *(.rodata*)         /* all read-only data sections from all files */
```

```
}

/* align at 4 KB and load at 1 MB + . */
.data ALIGN (0x1000) : AT(ADDR(.data)-0xC0000000)
{
    *(.data)            /* all data sections from all files */
}

/* align at 4 KB and load at 1 MB + . */
.bss ALIGN (0x1000) : AT(ADDR(.bss)-0xC0000000)
{
    *(COMMON)           /* all COMMON sections from all files */
    *(.bss)             /* all bss sections from all files */
}

kernel_virtual_end = .;
kernel_physical_end = . - 0xC0000000;
```

Assembly code can read these labels directly, and perhaps push them onto the stack so we can use them from C:

```
extern kernel_virtual_start
extern kernel_virtual_end
extern kernel_physical_start
extern kernel_physical_end

; ...

push kernel_physical_end
push kernel_physical_start
push kernel_virtual_end
push kernel_virtual_start

call kmain
```

This way we get the labels as arguments to `kmain`. If you want to use C instead of assembly, one way to do it is to declare the label as a function and take the address of the function:

```
void kernel_virtual_start(void);

/* ... */

unsigned int vaddr = (unsigned int) &kernel_virtual_start;
```

If you use GRUB modules you also need to make sure the memory they use is marked as "in use" as well.

Note that not all available memory needs to be contiguous. In the first 1MB there are several I/O-mapped memory sections, as well as memory used by GRUB and the BIOS. Other parts of the memory might be similarly unavailable.

It's convenient to divide the memory sections into complete page frames, as we can't map part of pages into memory.

Managing Available Memory

How do we know which page frames are in use? The page frame allocator needs to keep track of which are free and which aren't. There are several ways to do this: Bitmaps, linked lists, trees, the Buddy System (used by Linux) etc. See `http://wiki.osdev.org/Page_Frame_Allocation` for more details.

Bitmaps are quite easy to implement. One bit for each page frame, and dedicate one (or more) page frames to store the bitmap. But other designs might be better and/or more fun.

How Can We Access a Page Frame?

When we use our page frame allocator to allocate page frames, it gives us the physical start address of the page frame. This page frame is not mapped in - no page table points to the frame. (If we mapped an entire 4MB chunk for the kernel, and the page frame lies somewhere in there, it will be mapped in.) How can we read and write data to the frame?

We need to map the page frame into virtual memory, by updating the PDT and/or PT used by the kernel. What if all available page tables are full? Then we can't map the page frame into memory, because we'd need a new page table - which takes up an entire page frame - and to write to this page frame we'd need to map it in...

One solution is to reserve part of the first page table used by the kernel (or some other higher-half page table) for temporarily mapping in page frames so that we can access to them. If the kernel is mapped in at `0xC0000000` (page directory entry with index 768), and we've used 4KB page frames, the kernel has at least one page table. If we assume, or limit us to, a kernel of size at most 4MB minus 4KB, we can dedicate the last entry (entry 1023) of this page table for temporary mappings. The virtual address of pages mapped in like this will be:

```
(768 << 22) | (1023 << 12) | 0x000 = 0xC03FF000
```

After we've temporarily mapped in the page frame we want to use as a page table, and set it up to map in our first page frame, we can add it to the paging directory, and remove the temporary mapping. (This leads to the quite nice property that no paging tables need to be mapped in unless we need to edit them).

A Kernel Heap

So far we've been able to work with only fixed-size data, or directly with raw memory. Now that we have a page frame allocator we can implement `malloc` and `free` to use in the kernel.

Kernighan and Ritchie have an example implementation in [6] that we can draw inspiration from. The only real modifications we need to do is to remove calls to `sbrk`/`brk`, and directly ask the page frame allocator for the page frames, and the paging system to map them in.

This can be done quick and dirty. A real implementation should also give back page frames to the page frame allocator on `free`, whenever sufficiently large blocks are freed.

Further reading

- The OSDev wiki page on page frame allocation: `http://wiki.osdev.org/Page_Frame_Allocation`

Chapter 11

User Mode

User mode is now almost within our reach. There's just a few more steps required. They don't seem too much work on paper, but they can take quite a few hours to get right in code and design.

Segments for User Mode

To enable user mode we need to add two more segments to the GDT. They are very similar to the kernel segments we added when we set up the GDT in the chapter about segmentation.

Index	Offset	Name	Address range	Type	DPL
3	`0x18`	user code segment	`0x00000000 - 0xFFFFFFFF`	RX	PL3
4	`0x20`	user data segment	`0x00000000 - 0xFFFFFFFF`	RW	PL3

Table 11.1: The segment descriptors needed for user mode.

The difference is the DPL, which now allows code to execute in PL3. The segments can still be used to address the entire address space, so just using these segments for user mode code will not protect the kernel. For that we need paging.

Setting Up For User Mode

There are a few things every user mode process needs:

- Page frames for code, data and stack. For now we can just allocate one page frame for the stack, and enough page frames to fit the code and fixed-size data.
- We need to copy the binary from the GRUB modules to the frames used for code.
- We need a page directory and page tables to map these page frames into memory. At least two page tables are needed, because the code and data should be mapped in at `0x00000000` and increasing, and the stack should start just below the kernel, at `0xBFFFFFFB`, growing to lower addresses. Make sure the U/S flag is set to allow PL3 access.

It might be convenient to store these in a struct of some kind, dynamically allocated with the kernel `malloc`.

Entering User Mode

The only way to execute code with a lower privilege level than the current privilege level (CPL) is to execute an `iret` or `lret` instruction - interrupt return or long return, respectively.

To enter user mode, we set up the stack as if the processor had raised an inter-level interrupt. The stack should look like this:

```
[esp + 16]  ss      ; the stack segment selector we want for user mode
[esp + 12]  esp     ; the user mode stack pointer
[esp +  8]  eflags  ; the control flags we want to use in user mode
[esp +  4]  cs      ; the code segment selector
[esp +  0]  eip     ; the instruction pointer of user mode code to execute
```

(source: The Intel manual [28], section 6.2.1, figure 6-4)

`iret` will then read these values from the stack and fill in the corresponding registers. Before we execute `iret` we need to change to the page directory we setup for the user mode process. Important to remember here is that, to continue executing kernel code after we've switched PDT, the kernel needs to be mapped in. One way to accomplish this is to have a "kernel PDT", which maps in all kernel data at `0xC0000000` and above, and merge it with the user PDT (which only maps below `0xC0000000`) when we do the switch. Also, remember that we need to use the physical address for the page directory when we set `cr3`.

`eflags` is a register for a set of different flags, specified in section 2.3 of the Intel manual [28]. Most important for us is the interrupt enable (IF) flag. When in PL3 we aren't allowed to use `sti` like we'd normally do to enable interrupts. If interrupts are disabled when we enter user mode, we can't enable them when we get there. When we use `iret` to enter user mode it will set `eflags` for us, so setting the IF flag in the `eflags` entry on the stack will enable interrupts in user mode.

For now, we should have interrupts disabled, as it requires a little more twiddling to get inter-privilege level interrupts to work properly. See the section on system calls.

The `eip` should point to the entry point for the user code - `0x00000000` in our case. `esp` should be where the stack should start - `0xBFFFFFFB`.

`cs` and `ss` should be the segment selectors for the user code and user data segments, respectively. As we saw in the segmentation chapter, the lowest two bits of a segment selector is the RPL - the Requested Privilege Level. When we execute the `iret` we want to enter PL3, so the RPL of `cs` and `ss` should be 3.

```
cs = 0x18 | 0x3
ss = 0x20 | 0x3
```

`ds` - and the other data segment registers - should be set to the same segment selector as `ss`. They can be set the ordinary way, with `mov`.

Now we are ready to execute `iret`. If everything has been set up right, we now have a kernel that can enter user mode. Yay!

Using C for User Mode Programs

The kernel is compiled as an ELF [16] binary, which is a format that allows for more complex and convenient layouts and functionality than a plain flat binary. The reason we can use ELF for the kernel is that GRUB knows how to parse the ELF structure.

If we implement an ELF parser, we can compile the user mode programs into ELF binaries as well. We leave this as an exercise for the reader.

One thing we can do to make it easier to develop user mode programs is to allow them to be written in C but still compile them to flat binaries. In C the layout of the generated code is more unpredictable and the entry point (`main()`) might not be at offset 0.

One way to work around this is to add a few assembler lines placed at offset 0 which calls `main()`.

Assembler code (`start.s`):

```
extern main

section .text
    ; push argv
    ; push argc
    call main
    ; main has returned, eax is return value
    jmp  $    ; loop forever
```

Linker script (`link.ld`) to place `start.o` first:

```
OUTPUT_FORMAT("binary")    /* output flat binary */

SECTIONS
{
    . = 0;                 /* relocate to address 0 */

    .text ALIGN(4):
    {
        start.o(.text)     /* include the .text section of start.o */
        *(.text)           /* include all other .text sections */
    }

    .data ALIGN(4):
    {
        *(.data)
    }

    .rodata ALIGN(4):
    {
        *(.rodata*)
    }
}
```

Note: `*(.text)` will not include the `.text` section of `start.o` again. See [32] for more details.

With this script we can write programs in C or assembler (or any other language that compiles to object files linkable with `ld`), and it is easy to load and map in for the kernel. (`.rodata` will be mapped in as writeable, though.)

When we compile user programs we want the usual `CFLAGS`:

```
-m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector -nostartfiles
-nodefaultlibs
```

And these `LDFLAGS`:

```
-T link.ld -melf_i386  # emulate 32 bits ELF, the binary output is specified in the linker script
```

A C Library

It might now be interesting to start thinking about writing a short libc for your programs. Some of the functionality require system calls to work, but some, such as the `string.h` functions, does not.

Further Reading

- Gustavo Duarte has an article on privilege levels: `http://duartes.org/gustavo/blog/post/cpu-rings-privilege-and-protection`

Chapter 12

File Systems

We are not required to have file systems in our operating system, but it is quite convenient, and it often plays a central part in many operations of several existing OS's, especially UNIX-like systems. Before we start the process of supporting multiple processes and system calls, we might want to consider implementing a simple file system.

Why a File System?

How do we specify what programs to run in our OS? Which is the first program to run? How do programs output data? Read input?

In UNIX-like systems, with their almost-everything-is-a-file convention, these problems are solved by the file systems. It might also be interesting to read a bit about the Plan 9 project, which takes this idea one step further (See further reading below).

A Simple File System

The most simple file system possible might be what we already have - one "file", existing only in RAM, loaded by GRUB before the kernel starts. When our kernel and operating system grows, this is probably too limiting.

A next step could be to add structure to this GRUB-loaded module. With a utility program - perhaps called `mkfs` - which we run at build time, we can create our file system in this file.

`mkfs` can traverse a directory on our host system and add all subdirectories and files as part of our target file system. Each object in the file system (directory or file) can consist of a header and a body, where the body of a file is the actual file and the body of a directory is a list of entries - names and "addresses" of other files and directories.

Each object in this file system will become contiguous, so they will be easy to read from our kernel. All objects will also have a fixed size (except for the last one, which can grow); it might be difficult to add new or modify existing files. We can make the file system read-only.

`mmap` is a handy system call that makes writing the "file system-in-a-file" easier.

Inodes and Writable File Systems

When we decide that we want a more complex - and realistic - file system, we might want to look into the concept of inodes. See further reading.

A Virtual File System and devfs

What abstraction should we use for reading and writing to devices such as the screen and the keyboard?

With a virtual file system (VFS) we create an abstraction on top of any real file systems we might have. The VFS mainly supplies the path system and file hierarchy, and delegates operations on files to the underlying file systems. The original paper on VFS is succinct, concrete, and well worth a read. See further reading.

With a VFS we can mount a special file system on `/dev`, which handles all devices such as keyboards and the screen. Or we can take the traditional UNIX approach, with major/minor device numbers and `mknod` to create special files for our devices.

Further Reading

- The ideas behind the Plan 9 operating systems is worth taking a look at: `http://plan9.bell-labs.com/plan9/index.html`
- Wikipedia's page on inodes: `http://en.wikipedia.org/wiki/Inode` and the inode pointer structure: `http://en.wikipedia.org/wiki/Inode_pointer_structure`.
- The original paper on the concept of vnodes and a virtual file system is quite interesting: `http://www.arl.wustl.edu/~fredk/Courses/cs523/fall01/Papers/kleiman86vnodes.pdf`
- Poul-Henning Kamp discusses the idea of a special file system for `/dev` in `http://static.usenix.org/publications/library/proceedings/bsdcon02/full_papers/kamp/kamp_html/index.html`

Chapter 13

System Calls

What is a System Call?

System calls (syscalls) is the way for user-mode applications to interact with the kernel - to ask for resources, request operations to be performed, etc. The syscall API is the part of the kernel most exposed to the users, so its design requires some thought.

Designing System Calls

It is up to us, the kernel developers, to design the system calls that applications developer can use. We can draw inspiration from the POSIX standards or, if they seem a bit too much, just look at the ones for Linux, and pick and choose. See further reading below.

Implementing System Calls

Syscalls are traditionally implemented with software interrupts. The user applications puts the appropriate values in registers or on the stack and then initiates a pre-defined interrupt which transfers execution to the kernel. The interrupt number used is kernel-dependent. Linux uses `0x80`.

With "newer" versions of x86, Intel and AMD has implemented two different ways to initiate syscalls that are faster than interrupts - `syscall`/`sysenter` and `sysret`/`sysexit`, respectively.

When syscalls are executed, the current privilege level is typically changed from PL3 to PL0. To allow this, the DPL of the entry in the IDT for the syscall interrupt needs to allow PL3 access.

Whenever inter-privilege level interrupts occurs, the processor pushes a few important registers on the stack - the same ones we used to enter user mode before, see figure 6-4, section 6.12.1, in the Intel manual [28]. But what stack is used? The same section in [28] specifies that if the interrupt leads to code executing at numerically lower privilege levels, a stack switch occurs. The values for the new `ss` and `esp` is taken from the current Task State Segment (TSS). The TSS structure is specified in figure 7-2, section 7.2.1 of [28].

To enable syscalls we need to setup a TSS for our currently running programs before we enter user mode. Setting it up can easily be done in C, where we enter the `ss` and `esp` values. To load it into the processor, we need to set up a new TSS descriptor in our GDT. The structure of this descriptor is described in section 7.2.2 in [28].

We specify the current TSS segment selector by loading it into the `tr` register with `ltr`. If the TSS segment descriptor has index 5, and thus offset `5 * 8 = 40 = 0x28`, this is what we should load into `tr`.

When we entered user mode before we disabled interrupts when executing in PL3. For syscalls via interrupts to work, we need to enable interrupts. We enter user mode by using the `iret` instruction, and this loads the `eflags` register from the stack for us. Setting the IF flag bit in the `eflags` on the stack will make `iret` enable interrupts for us.

Further Reading

- The Wikipedia page on POSIX, with links to the specifications: `http://en.wikipedia.org/wiki/POSIX`
- A list of syscalls used in linux: `http://bluemaster.iu.hio.no/edu/dark/lin-asm/syscalls.html`
- The Wikipedia page on system calls: `http://en.wikipedia.org/wiki/System_call`
- The Intel manual [28] sections on interrupts (chapter 6) and TSS' (chapter 7) are where you get all the details you need.

Chapter 14

Scheduling

How do you make multiple processes appear to run at the same time? Today, this question has two answers:

- With the availability of multi-core processors, or on system with multiple processors, two processes can actually run at the same time by running two processes on different cores or processors
- Fake it. That is, switch rapidly (faster than a human can experience) between the processes. At any given moment, there is only process executing, but the rapid switching gives the impression that they are running "at the same time".

Since the operating system created in this book does not support multi-core processors or multiple processors, our only option is to fake it. The part of the operating system responsible for rapidly switching between the processes is called the *scheduler*.

Creating New Processes

Creating new processes is usually done with two different system calls: `fork` and `exec`. `fork` creates an exact copy of the currently running process, while `exec` replaces the current process with another, often specified by a path to the programs location in the file system. Of these two, we recommend that you start implementing `exec`, since this system call will do almost exactly the same steps as described in "Setting up for user mode".

Cooperative Scheduling with Yielding

The easiest way to achieve the rapid switching between processes is if the processes themselves are responsible for the switching. That is, the processes runs for a while and then tells the OS (via a syscall) that it can now switch to another process. Giving up the control of CPU to another process is called *yielding* and when the processes themselves are responsible for the scheduling, it's called *cooperative scheduling*, since all the processes must cooperate with each other.

When a process yields, the process entire state must be saved (all the registers), preferably in the kernel heap in a structure that represents a process. When changing to a new process, all the registers must be updated with the saved values.

In order to implement scheduling, you must keep a list of which processes are running. The system call "yield" should then run the next process in the list and puts the current one last (other schemes are possible, but this is a simple one). The transfer of control to the new process is done via `iret` in exactly the same way as explained in the section "Entering user mode".

We **strongly** recommend that you start to implement support for multiple processes by implementing cooperative scheduling. We further recommend that you have a working solution for both `exec`, `fork` and `yield` before implementing preemptive scheduling. The reason for this is because it is much easier to debug cooperative scheduling, since it is deterministic.

Preemptive Scheduling with the Timer

Instead of letting the processes themselves manage when it is time to change to another process, the OS can switch processes automatically after a short period of time. The OS can set up the programmable interval timer (PIT) to raise an interrupt after a short period of time, for example 20 ms. In the interrupt handler for the PIT interrupt, the OS will change the running process to a new one. This way, the processes themselves don't need to worry about scheduling.

Programmable Interval Timer

To be able to do preemptive scheduling, the programmable interval timer (PIT) must be first be configured to raise interrupts every x milliseconds, where x should be configurable.

The configuration of the PIT is very similar to the configuration of other hardware devices, you send a byte to an I/O port. The command port of the PIT is located at `0x43`. To read about all the configuration options, see [33]. We use the following options:

- Raise interrupts (use channel 0)
- Send the divider as low byte then high byte (see next section for an explanation)
- Use a square wave
- Use binary mode

This results in the configuration byte `00110110`.

Setting the interval for how often interrupts are to be raised is done via a *divider*, the same way as for the serial port. Instead of sending the PIT a value (e.g. in milliseconds) that says how often an interrupt should be raised, you send a *divider*. The PIT operates at 1193182 Hz as default. To change this, you send a divider, for example 10. The PIT will then run at `1193182 / 10 = 119318` Hz. The divider can only be 16 bits, so it only possible to configure the timers frequency between 1193182 Hz and `1193182 / 65535 = 18.2` Hz. We recommend that you create a function that takes an interval in milliseconds and converts it to the correct divider.

The divider is then sent to channel 0 data I/O port of the PIT, but since you can only send 1 byte at a time, you must first send the lowest 8 bits, then the highest 8 bits of the divider. The channel 0 data I/O port is at `0x40`. See [33] for more details.

Separate Kernel Stacks for Processes

If all processes uses the same kernel stack (the stack exposed by the TSS), there will be trouble if a process is interrupted while still in kernel mode. The process that is being switched to will now use the same kernel stack, and will then overwrite what the previous have written on the stack (remember that TSS data structure points to the *beginning* of the stack).

To solve this problem, every process should have it's own kernel stack, the same way that each process have their own user mode stack. When switching process, the TSS must then be updated to point to the new process kernel stack.

Difficulties with Preemptive Scheduling

When using preemptive scheduling, one problem arises that doesn't exist with cooperative scheduling. With cooperative scheduling, every time a process yields, it must be in user mode (privilege level 3), since yield is a system call. With preemptive scheduling, the processes can be interrupted in either user mode or kernel mode (privilege level 0), since the process itself does not control when it gets interrupted.

Interrupting a process in kernel mode is a little bit different than interrupting a process in user mode, due to the way the CPU sets up the stack at interrupts. If a privilege level change occurs (the process was interrupted in user mode), then the CPU will push the value of the process `ss` and `esp` register on the stack. If no privilege level change occurs (the process was interrupted in kernel mode), then the CPU won't push the `esp` register on the stack. Furthermore, if there is no privilege level change, the CPU won't change stack to the one defined it the TSS.

To fix this, you must yourself calculate what the value of `esp` was *before* the interrupt. Since you know that the CPU pushes 3 things on the stack when no privilege change happens and you know how much you have pushed on the stack, you can calculate what the value of `esp` was at the time of the interrupt. This is possible since the CPU won't change stacks if there is no privilege level change, so the content of `esp` will be the same as at the time of the interrupt.

To further complicate things, you must think of how to handle case when switching to a new process that should be running in kernel mode. Since you are then using `iret` without a privilege level change, the CPU won't update the value of `esp` with the one placed on the stack, you must update `esp` yourself.

Further Reading

- For more information about different scheduling algorithms, see `http://wiki.osdev.org/Scheduling_Algorithms`

Chapter 15

References

[1] Andrew Tanenbaum, 2007. *Modern Operating Systems, 3rd edition.* Prentice Hall, Inc.,

[2] OSDev, *OSDev*, http://wiki.osdev.org/Main_Page,

[3] James Molloy, *James M's kernel development tutorial*, http://www.jamesmolloy.co.uk/tutorial_html/,

[4] Canonical Ltd, *Ubuntu*, http://www.ubuntu.com/,

[5] Oracle, *Oracle VM VirtualBox*, http://www.virtualbox.org/,

[6] Dennis M. Ritchie Brian W. Kernighan, 1988. *The C Programming Language, Second Edition.* Prentice Hall, Inc.,

[7] Wikipedia, *C (programming language)*, http://en.wikipedia.org/wiki/C_(programming_language)/,

[8] Free Software Foundation, *GCC, the GNU Compiler Collection*, http://gcc.gnu.org/,

[9] NASM, *NASM: The Netwide Assembler*, http://www.nasm.us/,

[10] Wikipedia, *Shell script*, http://en.wikipedia.org/wiki/Shell_script/,

[11] Free Software Foundation, *GNU Make*, http://www.gnu.org/software/make/,

[12] Volker Ruppert, *bochs: The Open Souce IA-32 emulation project*, http://bochs.sourceforge.net/,

[13] QEMU, *QEMU*, http://wiki.qemu.org/Main_Page,

[14] Wikipedia, *BIOS*, https://en.wikipedia.org/wiki/BIOS,

[15] Free Software Foundation, *GNU GRUB*, http://www.gnu.org/software/grub/,

[16] Wikipedia, *Executable and Linkable Format*, http://en.wikipedia.org/wiki/Executable_and_Linkable_Format,

[17] Free Software Foundation, *Multiboot Specification version 0.6.96*, http://www.gnu.org/software/grub/manual/multiboot/

[18] Lars Nodeen, *Bug #426419: configure: error: GRUB requires a working absolute objcopy*, https://bugs.launchpad.net/ubuntu/+source/grub/+bug/426419,

[19] Wikipedia, *ISO image*, http://en.wikipedia.org/wiki/ISO_image,

[20] Bochs, *bochsrc*, http://bochs.sourceforge.net/doc/docbook/user/bochsrc.html,

[21] NASM, *RESB and Friends: Declaring Uninitialized Data*, http://www.nasm.us/doc/nasmdoc3.html#section-3.2.2,

[22] Wikipedia, *x86 calling conventions*, http://en.wikipedia.org/wiki/X86_calling_conventions,

[23] Wikipedia, *Framebuffer*, http://en.wikipedia.org/wiki/Framebuffer,

[24] Wikipedia, *ASCII*, https://en.wikipedia.org/wiki/Ascii,

[25] Wikipedia, *Serial port*, http://en.wikipedia.org/wiki/Serial_port,

[26] OSDev, *Serial Ports*, http://wiki.osdev.org/Serial_ports,

[27] WikiBooks, *Serial Programming/8250 UART Programming*, http://en.wikibooks.org/wiki/Serial_Programming/8250_UA

[28] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Vol. 3A*, http://www.intel.com/content/www/us/en/
and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html/,

[29] NASM, *Multi-Line Macros*, http://www.nasm.us/doc/nasmdoc4.html#section-4.3,

[30] SIGOPS, http://www.acm.uiuc.edu/sigops/roll_your_own/i386/irq.html,

[31] Andries Brouwer, *Keyboard scancodes*, http://www.win.tue.nl/∼,

[32] Steve Chamberlain, *Using ld, the GNU linker*, http://www.math.utah.edu/docs/info/ld_toc.html,

[33] OSDev, *Programmable Interval Timer*, http://wiki.osdev.org/Programmable_Interval_Timer,