# The little book about OS development

Erik Helin, Adam Renberg

# Contents

# Introduction

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

This book is not about the theory behind operating systems, or how any specific operating system (OS) works. For OS theory we recommend the book *Modern Operating Systems* by Andrew Tanenbaum [@ostanenbaum]. Lists and details on current operating systems are available on the Internet.

The starting chapters are 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.

In chapter 2 and 3 we set up our development environment and boot up our OS kernel in a virtual machine, eventually starting to write code in C. We continue in chapter 4 with writing to the screen and the serial port, and then we dive into segmentation in chapter 5 and interrupts and input in chapter 6.

After this we have a quite functional but bare-bones OS kernel. In chapter 7 we start the road to user mode applications, with virtual memory through paging (chapter 8 and 9), memory allocation (chapter 10), and finally running a user application in chapter 11.

In the last three chapters we discuss the more advanced topics of file systems (chapter 12), system calls (chapter 13), and multitasking (chapter 14).

## About the Book

The OS kernel and this book were produced as part of an advanced individual course at the Royal Institute of Technology [@kth], Stockholm. The authors had previously taken courses in OS theory, but had only minor practical experience with OS kernel development. In order to get more insight and a deeper understanding of how the theory from the previous OS courses works out in practice, the authors decided to create a new course, which focused on the development of a small OS. Another goal of the course was writing a thorough tutorial on how to develop a small OS basically from scratch, and this short book is the result.

The x86 architecture is, and has been for a long time, one of the most common hardware architectures. It was not a difficult choice to use the x86 architecture as the target of the OS, with its large community, extensive reference material and mature emulators. The documentation and information surrounding the details of the hardware we had to work with was not always easy to find or understand, despite (or perhaps due to) the age of the architecture.

The OS was developed in about six weeks of full-time work. The implementation was done in many small steps, and after each step the OS was tested manually. By developing in this incremental and iterative way, it was often easier to find any bugs that were introduced, since only a small part of the code had changed since the last known good state of the code. We encourage the reader to work in a similar way.

During the six weeks of development, almost every single line of code was written by the authors together (this way of working is also called *pair-programming*). It is our belief that we managed to avoid a lot of bugs due to this style of development, but this is hard to prove scientifically.

## The Reader

The reader of this book should be comfortable with UNIX/Linux, systems programming, the C language and computer systems in general (such as hexadecimal notation [@wiki:hex]). This book could be a way to get started learning those things, but it will be more difficult, and developing an operating system is already challenging on its own. Search engines and other tutorials are often helpful if you get stuck.

## Credits, Thanks and Acknowledgements

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

Most of the CSS formatting of the book is based on the work by Scott Chacon for the book Pro Git, https://git-scm.com/book/en/v2.

## Contributors

We are very grateful for the patches that people send us. The following users have all contributed to this book:

- alexschneider
- Avidanborisov
- nirs
- kedarmhaswade
- vamanea
- ansjob

## 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.

## Issues and where to get help

If you run into problems while reading the book, please check the issues on Github for help: https://github.com/OrdoFlammae/littleosbook-src/issues.

## License

however you want. References to this book are always received with warmth.

# First Steps

Developing an operating system (OS) is no easy task, and the question "How do I even begin to solve this problem?" is likely to come up several times during the course of the project for different problems. This chapter will help you set up your development environment and booting a very small (and primitive) operating system.

## Tools

### Quick Setup

We (the authors) have used Ubuntu [@ubuntu] as the operating system for doing OS development, running it both physically and virtually (using the virtual machine VirtualBox [@virtualbox]). A quick way to get everything up and running is to use the same setup as we did, since we know that these tools work with the samples provided in this book.

Once Ubuntu is installed, either physical or virtual, the following packages should be installed using `apt-get`:

```
sudo apt-get install build-essential nasm xorriso grub-pc-bin bochs bochs-sdl
```

### Programming Languages

The operating system will be developed using the C programming language [@knr][@wiki:c], using GCC [@gcc]. We use C because developing an OS requires a very precise control of the generated code and direct access to memory. Other languages that provide 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:

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

This attribute allows us to ensure that the compiler uses a memory layout for a `struct` exactly as we define it in the code. This is explained in more detail in the next chapter.

Due to this attribute, the example code might be hard to compile using a C compiler other than GCC.

For writing assembly code, we have chosen NASM [@nasm] as the assembler, since we prefer NASM's syntax over GNU Assembler.

Bash [@wiki:bash] will be used as the scripting language throughout the book.

### Host Operating System

All the code examples assumes that the code is being compiled on a UNIX like operating system. All code examples have been successfully compiled using Ubuntu [@ubuntu] versions 11.04 and 11.10.

### Build System

Make [@make] has been used when constructing the Makefile examples.

### Virtual Machine

When developing an OS it is very convenient to be able to run your code in a *virtual machine* instead of on a physical computer, since starting your OS in a virtual machine is much faster than getting your OS onto a physical medium and then running it on a physical machine. Bochs [@bochs] 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 [@qemu] and VirtualBox [@virtualbox]. This book uses Bochs.

By using a virtual machine we cannot ensure that our OS works on real, physical hardware. The environment simulated by the virtual machine is designed to be very similar to their physical counterparts, and the OS can be tested on one by just copying the executable to a CD and finding a suitable machine.

## Booting

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 example of the boot process:

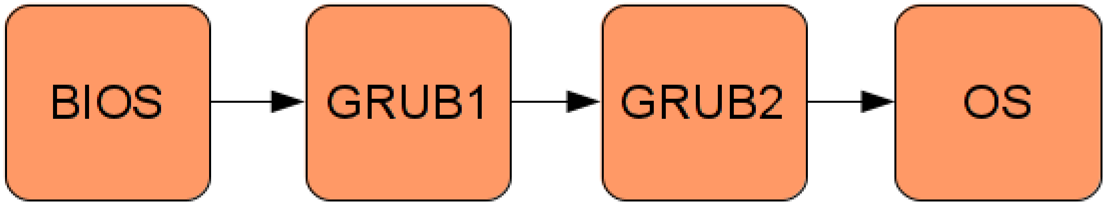

Figure 1: An example of the boot process. Each box is a program.

### BIOS

When the PC is turned on, the computer will start a small program that adheres to the *Basic Input Output System* (BIOS) [@wiki:bios] standard. This program is usually stored on a read only memory chip on the motherboard of the PC. The original role of the BIOS program was to export some library functions for printing to the screen, reading keyboard input etc. Modern operating systems do not use the BIOS' functions, they use drivers that interact directly with the hardware, bypassing the BIOS. 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 a *bootloader*. The bootloader's task is to transfer control to us, the operating system developers, and our code. However, due to some restrictions[1] of the hardware and because of backward compatibility, the bootloader is often split into two parts: the first

[1]The bootloader must fit into the *master boot record* (MBR) boot sector of a hard drive, which is only 512 bytes large.

part of the bootloader will transfer control to the second part, which finally gives control of the PC to the operating system.

Writing a bootloader involves writing a lot of low-level code that interacts with the BIOS. Therefore, an existing bootloader will be used: the GNU GRand Unified Bootloader (GRUB) [@grub].

Using GRUB, the operating system can be built as an ordinary ELF [@wiki:elf] executable, which will be loaded by GRUB into the correct memory location. The compilation of the kernel requires that the code is laid out in memory in a specific way (how to compile the kernel will be discussed later in this chapter).

### The Operating System

GRUB will transfer control to the operating system by jumping to a position in memory. Before the jump, GRUB will look for a magic number to ensure that it is actually jumping to an OS and not some random code. This magic number is part of the *multiboot specification* [@multiboot] which GRUB adheres to. Once GRUB has made the jump, the OS has full control of the computer.

## Hello Cafebabe

This section will describe how to implement of the smallest possible OS that can be used together with GRUB. The only thing the OS will do is write `0xCAFEBABE` to the `eax` register (most people would probably not even call this an OS).

### Compiling the Operating System

This part of the OS has to be written in assembly code, since C requires a stack, which isn't available (the chapter "Getting to C" describes how to set one up). Save the following code in a file called `loader.s`:

```
global loader                   ; the entry symbol for ELF

MAGIC_NUMBER equ 0x1BADB002     ; define the magic number constant
FLAGS        equ 0x0            ; multiboot flags
CHECKSUM     equ -MAGIC_NUMBER  ; calculate the checksum
                                ; (magic number + checksum + flags 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 to the machine code,
    dd FLAGS                    ; the flags,
    dd CHECKSUM                 ; and 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 OS will do is write the very specific number `0xCAFEBABE` to the `eax` register. It is *very* unlikely that the number `0xCAFEBABE` would be in the `eax` register if the OS did *not* put it there.

The file `loader.s` can be compiled into a 32 bits ELF [@wiki:elf] object file with the following command:

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

## 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. We want GRUB to load the kernel at a memory address larger than or equal to `0x00100000` (1 megabyte (MB)), because addresses lower than 1 MB are used by GRUB itself, BIOS and memory-mapped I/O. Therefore, the following linker script is needed (written for GNU LD [@gnubinutils]):

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

SECTIONS {
    . = 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 with the following command:

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

The final executable will be called `kernel.elf`.

## Building an ISO Image

The executable must be placed on a media that can be loaded by a virtual or physical machine. In this book we will use ISO [@wiki:iso] image files as the media, but one can also use floppy images, depending on what the virtual or physical machine supports.

We will create the kernel ISO image with the program `grub-mkrescue`. 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 the folder structure
    cp kernel.elf iso/boot/             # copy the kernel
```

A configuration file `grub.cfg` for GRUB must be created. This file tells GRUB where the kernel is located:

```
menuentry "os" {
    multiboot /boot/kernel.elf
}
```

Place the file `grub.cfg` in the folder `iso/boot/grub/`. The contents of the `iso` folder should now look like the following figure:

```
iso
|-- boot
  |-- grub
  | |-- grub.cfg
  |-- kernel.elf
```

The ISO image can then be generated with the following command:

```
grub-mkrescue -o os.iso iso
```

The `-o` flag specifies what to output the iso file to.

The ISO image `os.iso` now contains the kernel executable and the configuration file.

## Running Bochs

Now we can run the OS in the Bochs emulator using the `os.iso` ISO image. Bochs needs a configuration file to start and an example of a simple configuration file is given below:

```
megs:            32
display_library: sdl
romimage:        file=/usr/share/bochs/BIOS-bochs-latest
vgaromimage:     file=/usr/share/bochs/VGABIOS-lgpl-latest
ata0-master:     type=cdrom, path=os.iso, status=inserted
boot:            cdrom
log:             bochslog.txt
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. More information about the Bochs config file can be found at Boch's website [@bochs-config].

If you saved the configuration in a file named `bochsrc.txt` then you can run Bochs with the following command:

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

The flag `-f` tells Bochs to use the given configuration file and the flag `-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.

NB: On Debian and Ubuntu, the bochs package is compiled with the debugger. Because of this, you will have to go into the Bochs prompt, which will be in the terminal that initiated Bochs. Then type `c`, as a command. Then, Bochs will load your OS.

After quitting Bochs, display the log produced by Boch:

```
cat bochslog.txt
```

You should now see the contents of the registers of the CPU simulated by 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) in the output then your OS has successfully booted!

## Further Reading

- Gustavo Duertes has written an in-depth article about what actually happens when a x86 computer boots up, http://duartes.org/gustavo/blog/post/how-computers-boot-up
- Gustavo 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 an x86 computer: http://wiki.osdev.org/Boot_Sequence

# Getting to C

This chapter will show you how to use C instead of assembly code as the programming language for the OS. 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 use assembly code only where it makes sense.

## Setting Up a Stack

One prerequisite for using C is a stack, since all non-trivial C programs use 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 (remember that the stack grows towards lower addresses on the x86) that is correctly aligned (alignment on 4 bytes is recommended from a performance perspective).

We could point `esp` to a random 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 - we don't know how much memory is available or if the area `esp` would point to is used by something else. A better idea is to reserve a piece of uninitialized memory in the `bss` section in the ELF file of the kernel. It is better to use the `bss` section instead of the `data` section to reduce the size of the OS executable. Since GRUB understands ELF, GRUB will allocate any memory reserved in the `bss` section when loading the OS.

The NASM pseudo-instruction `resb` [@resb] can be used to declare uninitialized data:

```
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
```

There is no need to worry about the use of uninitialized memory for the stack, since it is not possible to read a stack location that has not been written (without manual pointer fiddling). A (correct) program can not pop an element from the stack without having pushed an element onto the stack first. Therefore, the memory locations of the stack will always be written to before they are being read.

The stack pointer is then set up by pointing `esp` to the end of the `kernel_stack` memory:

```
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 assembly code. There are many different conventions for how to call C code from assembly code [@wiki:ccall]. This book uses the *cdecl* calling convention, since that is 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 code shows an example:

```
/* The C function */
int sum_of_three(int arg1, int arg2, int arg3)
{
    return arg1 + arg2 + arg3;
}
```

```
; The assembly code
extern 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 this book, you will often come across “configuration bytes” that are a collection of bits in a very specific order. Below follows an example with 32 bits:

```
Bit:     | 31     24 | 23          8 | 7     0 |
Content: | index     | address       | config  |
```

Instead of using an unsigned integer, `unsigned int`, for handling such configurations, it is much more convenient to use “packed structures”:

```
struct example {
    unsigned char config;   /* bit 0 - 7   */
    unsigned short address; /* bit 8 - 23  */
    unsigned char index;    /* bit 24 - 31 */
};
```

When using the `struct` in the previous example there is no guarantee that the size of the `struct` will be exactly 32 bits - the compiler can add some padding between elements for various reasons, for example to speed up element access or due to requirements set by the hardware and/or compiler. When using a `struct` to represent configuration bytes, it is very important that the compiler does *not* add any padding, because the `struct` will eventually be treated as a 32 bit unsigned integer by the hardware. The attribute `packed` can be used to force GCC to *not* add any padding:

```
struct example {
    unsigned char config;   /* bit 0 - 7   */
    unsigned short address; /* bit 8 - 23  */
    unsigned char index;    /* bit 24 - 31 */
} __attribute__((packed));
```

Note that `__attribute__((packed))` is not part of the C standard - it might not work with all C compilers.

## Compiling C Code

When compiling the C code for the OS, a lot of flags to GCC need to be used. This is because the C code should *not* assume the presence of a standard library, since there is no standard library available for our OS.

For more information about the flags, see the GCC manual.

The flags used for compiling the C code 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 function `kmain` in a file called `kmain.c` that you call from `loader.s`. At this point, `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 test-run the OS. We recommend using `make` [@make], but there are plenty of other build systems available. A simple Makefile for the OS could look like the following example:

```
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
    grub-mkrescue -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
```

The contents of your working directory should now look like the following figure:

```
.
|-- bochsrc.txt
|-- iso
|   |-- boot
|     |-- grub
|       |-- grub.cfg
|-- kmain.c
|-- loader.s
|-- Makefile
```

You should now be able to start the OS with the simple command `make run`, which will compile the kernel and boot it up in Bochs (as defined in the Makefile above).

## Further Reading

- Kernigan & Richie's book, *The C Programming Language, Second Edition*, [@knr] is great for learning about all the aspects of C.

# Output

This chapter will present how to display text on the console as well as writing data to the serial port. Furthermore, we will create our first *driver*, that is, code that acts as a layer between the kernel and the hardware, providing a higher abstraction than communicating directly with the hardware. The first part of this chapter is about creating a driver for the *framebuffer* [@wiki:fb] to be able to display text on the console. The second part shows how to create a driver for the serial port. Bochs can store output from the serial port in a file, effectively creating a logging mechanism for the operating system.

## Interacting with the Hardware

There are usually two different ways to interact with 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 updated with the new data. One example of this is the framebuffer, which will be discussed in more detail later. For example, 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 code instructions `out` and `in` must be used to communicate with the hardware. The instruction `out` takes two parameters: the address of the I/O port and the data to send. The instruction `in` takes a single parameter, the address of the I/O port, and returns data from the hardware. One can think of I/O ports as communicating with 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 on a PC.

## The Framebuffer

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

### Writing Text

Writing text to the console via the framebuffer is done with memory-mapped I/O. The starting address of the memory-mapped I/O for the framebuffer is `0x000B8000` [@wiki:vga-compat]. The memory is divided into 16 bit cells, where the 16 bits determine both the character, the foreground color and the background color. The lowest eight bits is the ASCII [@wiki:ascii] value of the character, bit 8 - 11 the foreground and bit 12 - 14 the background and bit 15 blink, seen in the following figure:

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

The available colors are shown in the following table:

| 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 dark grey foreground (2) and green background (8) at place (0,0), the following assembly code instruction is used:

```
mov word [0x000B8000], 0x2841
```

The second cell then corresponds to row zero, column one and its address is therefore:

```
0x000B8000 + 2 = 0x000B8002
```

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

```
fb[0] = 'A';
fb[1] = 0x82;
```

The following code shows how this can be wrapped into a 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] = ((bg & 0x0F) << 4) | (fg & 0x0F)
}
```

The function can then be used as follows:

```
#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's position is determined with 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 code instruction argument is 8 bits, 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` [@osdev:vga] is the port that describes the data and port `0x3D5` [@osdev:vga] is for the data itself.

To set the cursor at row one, column zero (position `80 = 0x0050`), one would use the following assembly code 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 code instruction can't be executed directly in C. Therefore it is a good idea to wrap `out` in a function in assembly code which can be accessed from C via the cdecl calling standard [@wiki:ccall]:

```
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 code instruction can 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

The driver should provide an interface that the rest of the code in the OS will use for interacting with the framebuffer. There is no right or wrong in what functionality the interface should provide, but a suggestion is to have a `write` function with the following declaration:

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

The `write` function writes the contents of the buffer `buf` of length `len` to the screen. The `write` function should automatically advance the cursor after a character has been written and scroll the screen if necessary.

## The Serial Ports

The serial port [@wiki:serial] is an interface for communicating between hardware devices and although it is available on almost all motherboards, it is seldom exposed to the user in the form of a DE-9 connector nowadays. The serial port is easy to use, and, more importantly, it can be used as a logging utility in Bochs. If a computer has support for a serial port, then it usually has support for multiple serial ports, but we will only make use of one of the ports. This is because we will only use the serial ports for logging. Furthermore, we will only use the serial ports for output, not input. The serial ports are completely controlled via I/O ports.

### Configuring the Serial Port

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

- The speed used for sending data (bit or baud rate)
- If any error checking should be used for the data (parity bit, stop bits)
- The number of bits that represent 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 configuration.

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. We must therefore send an instruction telling the serial port to first expect the highest 8 bits, then the lowest 8 bits. This is done by sending `0x80` to the line command port. An example is shown below:

```
#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);
}
```

The way that data should be sent must be configured. This is also done via the line command port by sending a byte. The layout of the 8 bits looks like the following:

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

A description for each name can be found in the table below (and in [@osdev:serial]):

| 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 |
| s | The number of stop bits to use (`s = 0` equals 1, `s = 1` equals 1.5 or 2) |

| Name | Description |
|---|---|
| dl | Describes the length of the data |

We will use the mostly standard value `0x03` [@osdev:serial], meaning a length of 8 bits, no parity bit, one stop bit and break control disabled. This is sent to the line command port, as seen in the following example:

```
/** 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 | dl  |
     * Value:   | 0 | 0 | 0 0 0 | 0 | 1 1 | = 0x03
     */
    outb(SERIAL_LINE_COMMAND_PORT(com), 0x03);
}
```

The article on OSDev [@osdev:serial] has a more in-depth explanation of the values.

## Configuring the Buffers

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 data will be lost. In other words, the buffers are FIFO queues. The FIFO queue configuration byte looks like the following figure:

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

A description for each name can be found in the table below:

| Name | Description |
|---|---|
| lvl | How many bytes should be stored in the FIFO buffers |
| bs | If the buffers should be 16 or 64 bytes large |
| r | Reserved for future use |
| 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

The WikiBook on serial programming [@wikibook:serial] explains the values in more depth.

## 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 we want RTS and DTR to be 1, which means that we are ready to send data.

The modem configuration byte is shown in the following figure:

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

A description for each name can be found in the table below:

| 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 |

We don't need to enable interrupts, because we won't handle any received data. Therefore we use the configuration value `0x03 = 00000011` (RTS = 1 and DTS = 1).

## Writing Data to the Serial Port

Writing data to the serial port is done via the data I/O port. However, before writing, the transmit FIFO queue has to be empty. The line status I/O register uses bit 6 to denote that all previous writes have been completed, but for us it is sufficient to simply know that the transmit FIFO queue is empty. This is the case if bit 5 of the line status I/O port is equal to one.

Reading the contents of an I/O port is done via the `in` assembly code instruction. There is no way to use the `in` assembly code instruction from C, therefore it has to be wrapped (the same way as the `out` assembly code 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 then 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 queue is empty
 */
int serial_is_transmit_fifo_empty(unsigned int com)
{
    /* 0x20 = 0010 0000 */
    return inb(SERIAL_LINE_STATUS_PORT(com)) & 0x20;
}
```

Writing to a serial port means spinning as long as the transmit FIFO queue isn't empty, and then writing the data to the data I/O port.

### Configuring Bochs

To save the output from the first serial port the Bochs configuration file `bochsrc.txt` must be updated. The `com1` configuration instructs Bochs how to handle first serial port:

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

The output from serial port one will now be stored in the file `com1.out`.

### The Driver

We recommend that you implement a `write` function for the serial port similar to the `write` function in the driver for the framebuffer. To avoid name clashes with the `write` function for the framebuffer it is a good idea to name the functions `fb_write` and `serial_write` to distinguish them.

We further recommend that you try to write a `printf`-like function, see section 7.3 in [@knr]. The `printf` function could take an additional argument to decide to which device to write the output (framebuffer or serial).

A final recommendation is that you create some way of distinguishing the severeness of the log messages, 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

# 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 that specifies 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-rights checks ignored for now) the result is a *linear address*. When paging is disabled, then the linear address space is mapped 1:1 onto the *physical address* space, and the physical memory can be accessed. (See the chapter “Paging” for how to enable paging.)

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 all processes have their own LDT. LDTs can be used if a more complex segmentation model is desired - we won't use it. The GDT is shared by everyone - it's global.

As we discuss in the sections on virtual memory and paging, segmentation is rarely used more than in a minimal setup, similar to what we do below.

## Accessing Memory

Most of the time when accessing memory there is no need to explicitly specify the segment to use. The processor has six 16-bit segment registers: `cs`, `ss`, `ds`, `es`, `gs` and `fs`. The register `cs` is the code segment register and specifies the segment to use when fetching instructions. The register `ss` is used whenever accessing the stack (through the stack pointer `esp`), and `ds` is used for other data accesses. The OS is free to use the registers `es`, `gs` and `fs` however it want.

Below is an example showing implicit use of the segment registers:

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

The above example can be compared with the following one that makes explicit use of the segment registers:

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

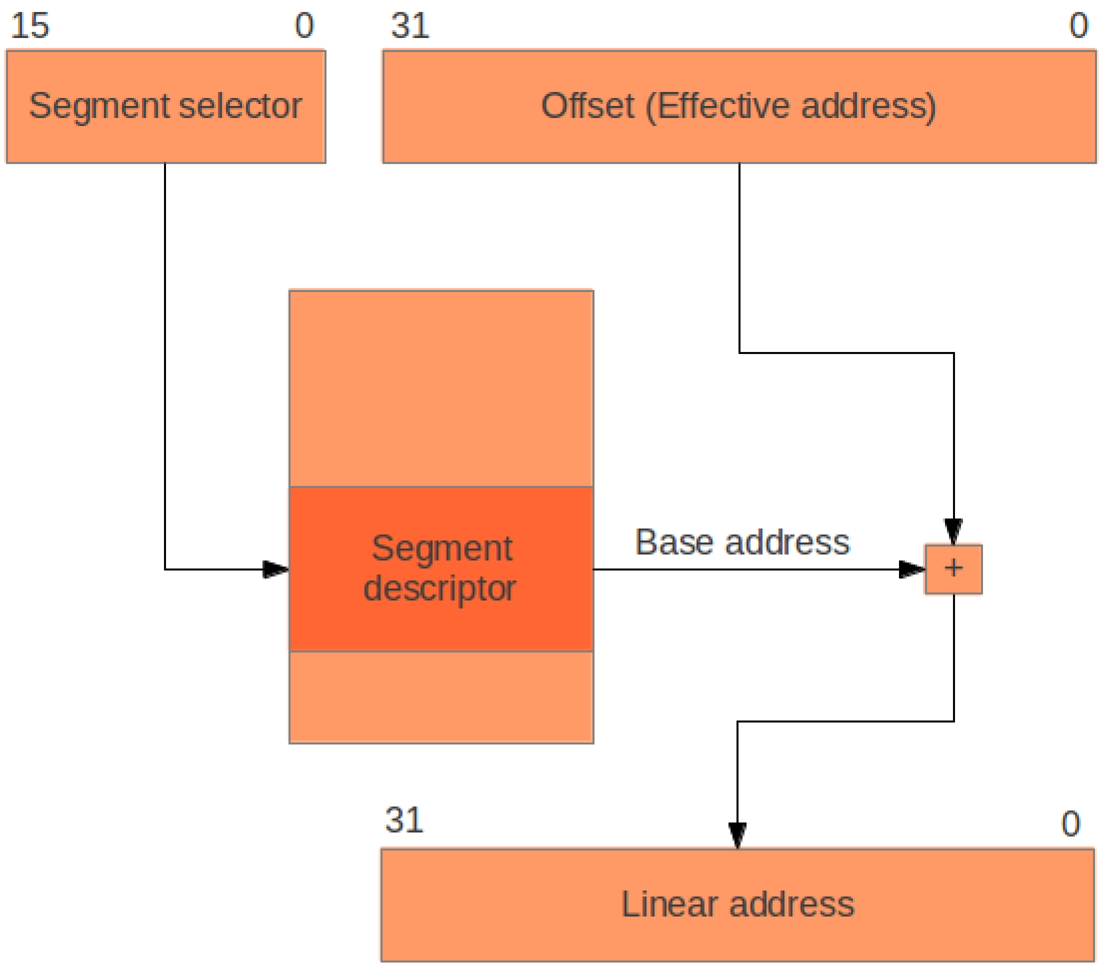

Figure 2: Translation of logical addresses to linear addresses.

```
    ret
```

You don't need to use `ss` for storing the stack segment selector, or `ds` for the data segment selector. You could store the stack segment selector in `ds` and vice versa. However, in order to use the implicit style shown above, you must store the segment selectors in their indented registers.

Segment descriptors and their fields are described in figure 3-8 in the Intel manual [@intel3a].

## 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. At least two segment descriptors (plus the null descriptor) are needed for the GDT, because the descriptor contains more information than just the base and limit fields. The two most relevant fields for us are the *Type* field and the *Descriptor Privilege Level* (DPL) field.

Table 3-1 in chapter 3 of the Intel manual [@intel3a] specifies the values for the Type field. The table shows that the Type field can't be both writable *and* executable at the same time. Therefore, two segments are needed: one segment for executing code to put in `cs` (Type is Execute-only or Execute-Read) and one segment for reading and writing data (Type is Read/Write) to put in the other segment registers.

The DPL specifies the *privilege levels* required to use the 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. However, some operating system, such as MINIX, make use of all levels. The kernel should be able to do anything, therefore it uses segments with DPL set to 0 (also called kernel mode). The current privilege level (CPL) is determined by the segment selector in `cs`.

The segments needed are described in the table below.

Table 5: The segment descriptors needed.

| 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 |

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 Intel manual [@intel3a], chapter 3, for details on the other descriptor fields.

## Loading the GDT

Loading the GDT into the processor is done with the `lgdt` assembly code instruction, which takes the address of a struct that specifies the start and size of the GDT. It is easiest to encode this information using a "packed struct" as shown in the following example:

```
struct gdt {
    unsigned short size;
    unsigned int address;
} __attribute__((packed));
```

If the content of the `eax` register is the address to such a struct, then the GDT can be loaded with the assembly code shown below:

```
lgdt [eax]
```

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

After the GDT has been loaded the segment registers needs to be loaded with their corresponding segment selectors. The content of a segment selector is described in the figure and table below:

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

Table 6: The layout of segment selectors.

| 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. |

The offset of the segment selector 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 the kernel of the OS should execute in privilege level 0.

Loading the segment selector registers is easy for the data registers - just copy the correct offsets to the registers. However, there is one caveat. In order to assign values to these registers, you cannot move them directly. Instead, you need to assign values to a general-purpose register (ex. `eax`) first, then move from there into the segment selector register:

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

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

```
; code here uses the previous cs
jmp 0x08:flush_cs   ; specify cs when jumping to 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.

## Further Reading

- Chapter 3 of the Intel manual [@intel3a] is filled with low-level and technical details about segmentation.
- The OSDev wiki has a page about segmentation: http://wiki.osdev.org/Segmentation
- The Wikipedia page on x86 segmentation might be worth looking into: http://en.wikipedia.org/wiki/X86_memory_segmentation

# Interrupts and Input

Now that the OS can produce *output* it would be nice if it also could get some *input*. (The operating system must be able to handle *interrupts* in order to read information from the keyboard). 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 have access to, or when a program divides a number by zero. Finally, there are also *software interrupts*, which are interrupts that are caused by the `int` assembly code instruction, and they are often used for system calls.

## Interrupts Handlers

Interrupts are handled via the *Interrupt Descriptor Table* (IDT). The IDT describes a handler for each interrupt. The interrupts are numbered (0 - 255) and the handler for interrupt *i* is defined at the *ith* position in the table. There are three different kinds of handlers for interrupts:

- Task handler
- Interrupt handler
- Trap handler

The task handlers use functionality specific to the Intel version of x86, so they won't be covered here (see the Intel manual [@intel3a], chapter 6, for more info). The only difference between an interrupt handler and a trap handler is that the interrupt handler disables interrupts, which means you cannot get an interrupt while at the same time handling an interrupt. In this book, we will use trap handlers and disable 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 shown in the figure below:

```
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  |
```

The lowest 32 bits are presented in the following figure:

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

A description for each name can be found in the table below:

| Name | Description |
|---:|---|
| offset high | The 16 highest bits of the 32 bit address in the segment. |

| Name | Description |
|---|---|
| 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 offset in the GDT. |
| r | Reserved. |

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

```
0xDEAD8E00
0x0008BEEF
```

If the IDT is represented as an `unsigned int idt[512]` then to register the above example as an handler for interrupt 0 (divide-by-zero), the following code would be used:

```
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.

## Handling an Interrupt

When an interrupt occurs the CPU will push some information about the interrupt onto the stack, then look up the appropriate interrupt hander in the IDT and jump to it. The stack at the time of the interrupt will look like the following:

```
[esp + 12] eflags
[esp + 8]  cs
[esp + 4]  eip
[esp]      error code?
```

The reason for the question mark behind error code is that not all interrupts create an *error code*. The specific CPU interrupts that put an error code on the stack are 8, 10, 11, 12, 13, 14 and 17. The error code can be used by the interrupt handler to get more information on what has happened. Also, note that the interrupt *number* is *not* pushed onto the stack. We can only determine what interrupt has occurred by knowing what code is executing - if the handler registered for interrupt 17 is executing, then interrupt 17 has occurred.

Once the interrupt handler is done, it uses the `iret` instruction to return. The instruction `iret` expects the stack to be the same as at the time of the interrupt (see the figure above). Therefore, any values pushed onto the stack by the interrupt handler must be popped. Before returning, `iret` restores `eflags` by popping the value from the stack and then finally jumps to `cs:eip` as specified by the values on the stack.

The interrupt handler has to be written in assembly code, since all registers that the interrupt handlers use must be preserved by pushing them onto the stack. This is because the code that was interrupted doesn't know about the interrupt and will therefore expect that its registers stay the same. Writing all the logic of the interrupt handler in assembly code will be tiresome. Creating a handler in assembly code that saves the registers, calls a C function, restores the registers and finally executes `iret` 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 as arguments. 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, struct stack_state stack, unsigned int interrupt);
```

## Creating a Generic Interrupt Handler

Since the CPU does not push the interrupt number on the stack it is a little tricky to write a generic interrupt handler. This section will use macros to show how it can be done. Writing one version for each interrupt is tedious - it is better to use the macro functionality of NASM [@nasm:macros]. And since not all interrupts produce an error code the value 0 will be added as the "error code" for interrupts without an error code. The following code shows an example of how this can be done:

```
%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 on the stack.
- Call the C function `interrupt_handler`.
- Pop the registers from the stack.
- Add 8 to `esp` (because of the error code and the interrupt number pushed earlier).
- Execute `iret` to return to the interrupted code.

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

## Loading the IDT

The IDT is loaded with the `lidt` assembly code instruction which takes the address of the first element in the table. It is easiest to wrap this instruction and 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. The PIC uses interrupts 0 - 15 for hardware interrupts by default, which conflicts with the CPU interrupts. Therefore the PIC interrupts must be remapped to another interval.
- Select which interrupts to receive. You probably don't want to receive interrupts from all devices since you don't have code that handles 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 were added, 8 interrupts were too few. The solution chosen was to chain on another PIC (PIC 2) on the first PIC (see interrupt 2 on PIC 1).

The hardware interrupts are shown in the table below:

| 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 the SigOPS website [@acm]. We won't repeat that information here.

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.

Acknowledging a PIC interrupt is done by sending the byte `0x20` to the PIC that raised the interrupt. Implementing a `pic_acknowledge` function can thus be done as follows:

```
#include "io.h"

#define PIC1_PORT_A 0x20
#define PIC2_PORT_A 0xA0

/* The PIC interrupts have been remapped */
#define PIC1_START_INTERRUPT 0x20
#define PIC2_START_INTERRUPT 0x28
#define PIC2_END_INTERRUPT   PIC2_START_INTERRUPT + 7

#define PIC_ACK     0x20

/** pic_acknowledge:
 *  Acknowledges an interrupt from either PIC 1 or PIC 2.
 *
 *  @param num The number of the interrupt
 */
```

```
void pic_acknowledge(unsigned int interrupt)
{
    if (interrupt < PIC1_START_INTERRUPT || interrupt > PIC2_END_INTERRUPT) {
      return;
    }

    if (interrupt < PIC2_START_INTERRUPT) {
      outb(PIC1_PORT_A, PIC_ACK);
    } else {
      outb(PIC2_PORT_A, PIC_ACK);
    }
}
```

## Reading Input from the Keyboard

The keyboard does not generate ASCII characters, it generates scan codes. A scan code represents a button - both presses and releases. The scan code representing the just pressed button can be read from the keyboard's data I/O port which has address `0x60`. How this can be done is shown in the following example:

```
#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 ASCII characters as is done on an American keyboard then Andries Brouwer has a great tutorial [@scancodes].

Remember, since the keyboard interrupt is raised by the PIC, you must call `pic_acknowledge` at the end of the keyboard interrupt handler. Also, the keyboard will not send you any more interrupts until you read the scan code from the keyboard.

## Further Reading

- The OSDev wiki has a great page on interrupts, http://wiki.osdev.org/Interrupts
- Chapter 6 of Intel Manual 3a [@intel3a] describes everything there is to know about interrupts.

# 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) to make application development 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. This environment 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 the OS created in this book can execute programs in user mode, but this chapter will show how to easily execute a small program in kernel mode.

## Loading an External Program

Where do we get the external program from? Somehow we need to load the code we want to execute into memory. More feature-complete operating systems usually have drivers and file systems that enable them to load the software from a CD-ROM drive, a hard disk or other persistent media.

Instead of creating all these drivers and file systems we will use a feature in GRUB called modules to load the 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
```

The application `program` will be created later in this chapter.

The code that calls `kmain` must be updated 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 "Paging" for details about page alignment).

To instruct GRUB how to load our modules, the "multiboot header" - 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 it an argument for `kmain`.

## Executing a Program

### A Very Simple Program

A program written at this stage can only perform a few actions. Therefore, a very short program that writes a value to a register suffices as a test program. Halting Bochs after a while and then check that register contains the correct number by looking in the Bochs log will verify that the program has run. This is an example of such a short program:

```
; 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 this 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 `ebx` is passed as an argument to `kmain`, we can do this entirely from C.

The pointer in `ebx` points to a *multiboot* structure [@multiboot]. 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` in the `ebx` register can be cast to a `multiboot_info_t` pointer. The address of the first module is in the field `mods_addr`. The following code shows an example:

```
int kmain(/* additional arguments */ 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 by GRUB. This can be done by checking the `flags` field of the `multiboot_info_t` structure. You should also check the field `mods_count` to make sure it is exactly 1. For more details about the multiboot structure, see the multiboot documentation [@multiboot].

### Jumping to the Code

The only thing left to do is to jump to the code loaded by GRUB. Since it is easier to parse the multiboot structure in C than assembly code, calling the code from C is more convenient (it can of course be done with `jmp` or `call` in assembly code as well). The C code could look like this:

```
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 the register `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 at the same privilege level 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, beside *segmentation*, do *paging* and *page frame allocation*.

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

# A Short Introduction to Virtual Memory

*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 in the machine. We also don't want applications messing with the kernel or other applications' memory due to security.

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 the next chapter. Some use of segmentation is still necessary to allow for code to execute under different privilege levels.

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 [@intel3a], 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. This way no process can see the memory of another process. A problem with this is that the physical memory for a process needs to be contiguous (or at least it is very convenient if it is). 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). Paging solves both these problems.

It is interesting to note that in x86_64 (the 64-bit version of the x86 architecture), segmentation is almost completely removed.

## Further Reading

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

# 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 through paging means that each process will get the impression that the available memory range is `0x00000000 - 0xFFFFFFFF` even though the actual size of the memory might be much less. It also means that when a process addresses a byte of memory it 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 gets 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 do not make use of it. But if we want to mark certain areas of memory accessible only to code running at a certain privilege level (to be able to have processes running at different privilege levels), paging is the neatest way to do it.

## Paging in x86

Paging in x86 (chapter 4 in the Intel manual [@intel3a]) 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 (linear) address, the highest 10 bits specifies the offset of a page directory entry (PDE) in the current PDT, the next 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 times 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.

While pages are normally 4096 bytes, it is also possible to use 4 MB pages. A PDE then points directly to a 4 MB page frame, which needs to be aligned on a 4 MB 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 4 MB and 4 KB pages.

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

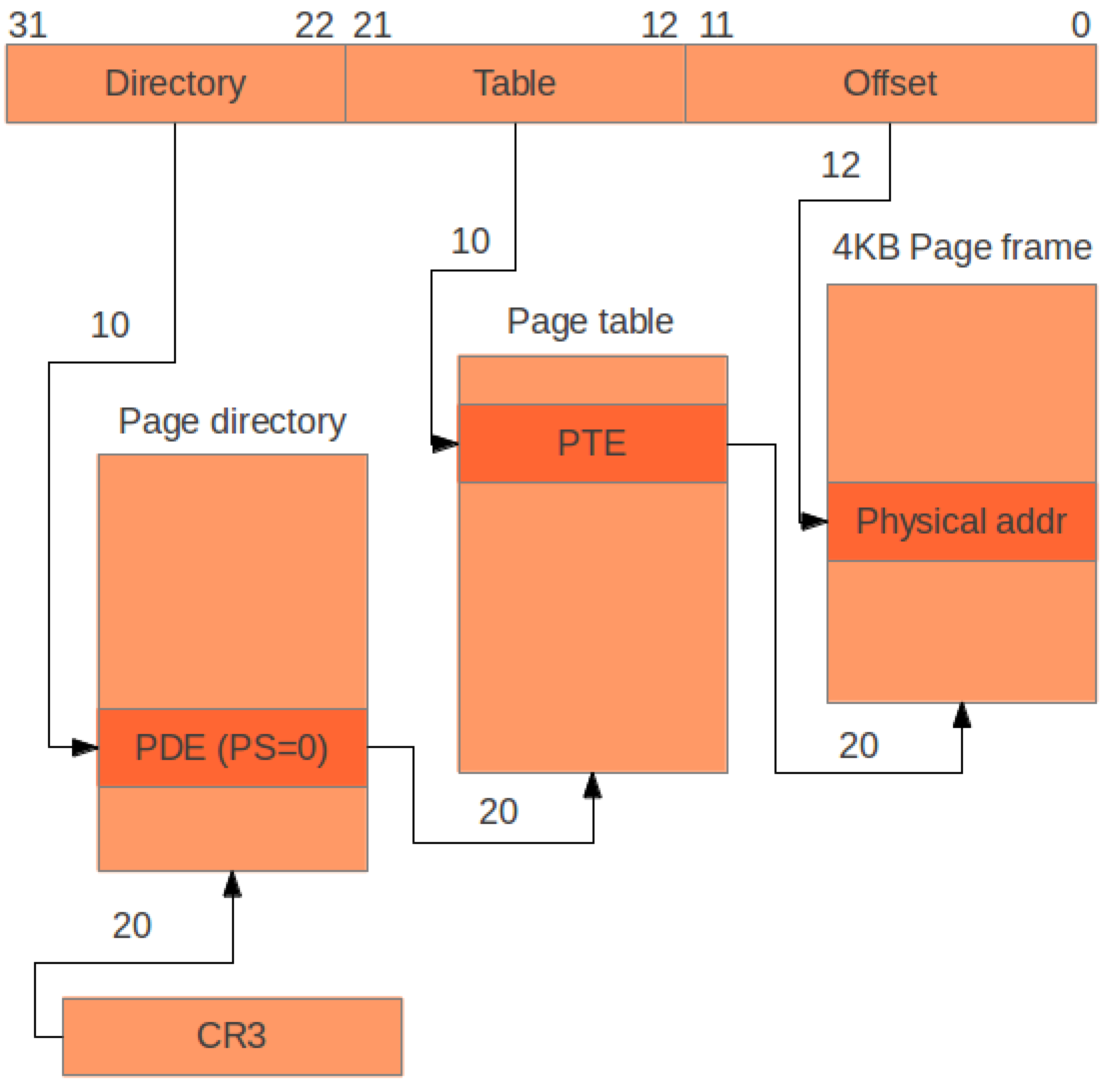

Figure 3: Translating virtual addresses (linear addresses) to physical addresses.

For more details on the paging structures, see chapter 4 in the Intel manual [@intel3a]. 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, called *identity paging*. This can be done at compile time by creating a page directory where each entry points to its corresponding 4 MB frame. In NASM this can be done with macros and commands (`%rep`, `times` and `dd`). It can of course also be done at run-time by using ordinary assembly code instructions.

### Enabling Paging

Paging is enabled 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`. To use 4 MB pages, set the PSE bit (Page Size Extensions, bit 4) of `cr4`. The following assembly code shows an example:

```
; 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` need to be physical addresses to the structures, never virtual. This will be more relevant in later sections where we dynamically update the paging structures (see the chapter "User Mode").

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

An example of invalidating a TLB entry is shown below:

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

## Paging and the Kernel

This section will describe how paging affects the OS kernel. We encourage you to run your OS using identity paging before trying to implement a more advanced paging setup, since it can be hard to debug a malfunctioning page table that is set up via assembly code.

### 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 space (`0x00000000`, `"size of kernel"`) maps 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 kernel's 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 3 GB large, which is now the only way that it can conflict with the kernel. When the kernel uses virtual addresses at 3 GB and above it is called a *higher-half kernel*. `0xC0000000` is just an example, the kernel can be placed at any address higher than 0 to get the same benefits. Choosing the correct address depends on how much virtual memory should be available for the kernel (it is easiest if all memory above the kernel virtual address should belong to the kernel) and how much virtual memory should be available for the process.

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

### Placing the Kernel at `0xC0000000`

To start with, it is better to place the kernel at `0xC0100000` than `0xC0000000`, since this makes it possible to map (`0x00000000`, `0x00100000`) to (`0xC0000000`, `0xC0100000`). This way, the entire range (`0x00000000`, `"size of kernel"`) of memory is mapped to the range (`0xC0000000`, `0xC0000000  + "size of kernel"`).

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 relocation is used in the linker script (see the section "Linking the kernel"). However, we want the jumps to be resolved using `0xC0100000` as base address, 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`).

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. Furthermore, we 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 GNU ld [@ldcmdlang], the load address specified by `AT` is handled by GRUB when loading the kernel, and is part of the ELF format [@wiki:elf].

## 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 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(.text)-0xC0000000)
{
    *(.rodata*)         /* all read-only data sections from all files */
}

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

/* align at 4 KB and load at 1 MB + . */
.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 computer has more than 3 GB of memory) the computer will just crash.

Therefore, assembly code that doesn't use relative jumps or relative memory addressing must be used to do the following:

- Set up a page table.
- Add identity mapping for the first 4 MB of the virtual address space.
- Add 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 jump can be done to a label to make `eip` point to a virtual address in the higher half:

```
; 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.
```

The register `eip` will now point to a memory location somewhere right after `0xC0100000` - all the code can now execute as if it were located at `0xC0100000`, the higher-half. The entry mapping of the first 4 MB of virtual memory to the first 4 MB of physical memory can now 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 more details we must deal with when using a higher-half kernel. We must be careful 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 4 MB pages for the kernel is simple, but wastes memory (unless you have a really big kernel). Creating a higher-half kernel mapped in as 4 KB pages saves memory but is harder to set up. Memory for the page directory and one page table can be reserved in the `.data` section, but one needs to configure the mappings from virtual to physical addresses at run-time. The size of the kernel can be determined by exporting labels from the linker script [@ldcmdlang], which we'll need to do later anyway when writing the page frame allocator (see the chapter "Page Frame Allocation).

## 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 data around in memory. We can also allow the user mode programs access to all memory below 3 GB, 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 [@intel3a] 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 at Steve Chamberlain's website [@ldcmdlang].
- More details on the ELF format can be found in this presentation: http://flint.cs.yale.edu/cs422/doc/ELF_Format.pdf

# Page Frame Allocation

When using virtual memory, how does the OS know which parts of memory are free to use? That is the role of the page frame allocator.

## Managing Available Memory

### How Much Memory is There?

First we need to know how much memory is available on the computer the OS is running on. The easiest way to do this is to read it from the multiboot structure [@multiboot] passed to us by GRUB. GRUB collects the information we need about the memory - what is reserved, I/O mapped, read-only etc. We must also make sure that we don't mark the part of memory used by the kernel as free (since GRUB doesn't mark this memory as reserved). One way to know how much memory the kernel uses is to export labels at the beginning and the end of the kernel binary from the 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;
```

These labels can directly be read from assembly code and pushed on the stack to make them available to C code:

```
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 code, one way to do it is to declare the labels as functions and take the addresses of these functions:

```
void kernel_virtual_start(void);

/* ... */

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

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

Note that the available memory does not need to be contiguous. In the first 1 MB 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. For more information about the different algorithms see the article on OSDev [@osdev:pfa].

Bitmaps are quite easy to implement. One bit is used for each page frame and one (or more) page frames are dedicated to store the bitmap. (Note that this is just one way to do it, other designs might be better and/or more fun to implement.)

## How Can We Access a Page Frame?

The page frame allocator returns the physical start address of the page frame. This page frame is not mapped in - no page table points to this page frame. 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 its page frame… Somehow this circular dependency must be broken.

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

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

After we've temporarily mapped 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.

## A Kernel Heap

So far we've only been able to work with 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 [@knr] have an example implementation in their book [@knr] that we can draw inspiration from. The only modification we need to do is to replace calls to `sbrk`/`brk` with calls to the page frame allocator when more memory is needed. We must also make sure to map the page frames returned by the page frame allocator to virtual addresses. A correct implementation should also return page frames to the page frame allocator on call to `free`, whenever sufficiently large blocks of memory are freed.

## Further reading

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

# User Mode

User mode is now almost within our reach, there are just a few more steps required to get there. Although these steps might seem easy the way they are presented in this chapter, they can be tricky to implement, since there are a lot of places where small errors will cause bugs that are hard to find.

## 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:

Table 9: The segment descriptors needed for user mode.

| 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 |

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, 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. At the moment it suffices to allocate one page frame for the stack and enough page frames to fit the program's code. Don't worry about setting up a stack that can grow and shrink at this point in time, focus on getting a basic implementation work first.
- The binary from the GRUB module has to be copied to the page frames used for the programs code.
- A page directory and page tables are needed to map the page frames described above 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 towards lower addresses. The U/S flag has to be set to allow PL3 access.

It might be convenient to store this information in a `struct` representing a process. This process `struct` can be dynamically allocated with the kernel's `malloc` function.

## 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-privilege level interrupt. The stack should look like the following:

```
[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
```

See the Intel manual [@intel3a], section 6.2.1, figure 6-4 for more information.

The instruction `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. It is important to remember 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 separate PDT for the kernel, which maps all data at `0xC0000000` and above, and merge it with the user PDT (which only maps below `0xC0000000`) when performing the switch. Remember that physical address of the PDT has to be used when setting the register `cr3`.

The register `eflags` contains a set of different flags, specified in section 2.3 of the Intel manual [@intel3a]. Most important for us is the interrupt enable (IF) flag. The assembly code instruction `sti` can't be used in privilege level 3 for enabling interrupts. If interrupts are disabled when entering user mode, then interrupts can't enabled once user mode is entered. Setting the IF flag in the `eflags` entry on the stack will enable interrupts in user mode, since the assembly code instruction `iret` will set the register `eflags` to the corresponding value on the stack.

For now, we should have interrupts disabled, as it requires a little more work to get inter-privilege level interrupts to work properly (see the section "System calls").

The value `eip` on the stack should point to the entry point for the user code - `0x00000000` in our case. The value `esp` on the stack should be where the stack starts - `0xBFFFFFFB` (`0xC0000000 - 4`).

The values `cs` and `ss` on the stack 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 using `iret` to enter PL3, the RPL of `cs` and `ss` should be `0x3`. The following code shows an example:

```
USER_MODE_CODE_SEGMENT_SELECTOR equ 0x18
USER_MODE_DATA_SEGMENT_SELECTOR equ 0x20
mov cs, USER_MODE_CODE_SEGMENT_SELECTOR | 0x3
mov ss, USER_MODE_DATA_SEGMENT_SELECTOR | 0x3
```

The register `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 the `mov` assembly code instruction.

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

## Using C for User Mode Programs

When C is used as the programming language for user mode programs, it is important to think about the structure of the file that will be the result of the compilation.

The reason we can use ELF [@wiki:elf] as the file format for for the kernel executable is because GRUB knows how to parse and interpret the ELF file format. If we implemented an ELF parser, we could 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 the programs to be written in C, but compile them to flat binaries instead of ELF binaries. In C the layout of the generated code is more unpredictable and the entry point, `main`, might not be at offset 0 in the binary. One common way to work around this is to add a few assembly code lines placed at offset 0 which calls `main`:

```
extern main

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

If this code is saved in a file called `start.s`, then the following code show an example of a linker script that places these instructions first in executable (remember that `start.s` gets compiled to `start.o`):

```
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.

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 for the kernel (`.rodata` will be mapped in as writeable, though).

When we compile user programs we want the following GCC flags:

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

For linking, the followings flags should be used:

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

The option `-T` instructs the linker to use the linker script `link.ld`.

### A C Library

It might now be interesting to start thinking about writing a small “standard library” for your programs. Some of the functionality requires system calls to work, but some, such as the functions in `string.h`, does not.

## Further Reading

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

# File Systems

We are not required to have file systems in our operating system, but it is a very usable abstraction, and it often plays a central part of many operating systems, especially UNIX-like operating 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 or read input?

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

## A Simple Read-Only File System

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

A file system that is slightly more advanced than just the bits of one file is a file with metadata. The metadata can describe the type of the file, the size of the file and so on. A utility program can be created that runs at build time, adding this metadata to a file. This way, a "file system in a file" can be constructed by concatenating several files with metadata into one large file. The result of this technique is a read-only file system that resides in memory (once GRUB has loaded the file).

The program constructing the file system can traverse a directory on the host system and add all subdirectories and files as part of the 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 memory for the kernel. All objects will also have a fixed size (except for the last one, which can grow), therefore it is difficult to add new files or modify existing ones.

## Inodes and Writable File Systems

When the need for a writable file system arises, then it is a good idea to look into the concept of an *inode*. See the section "Further Reading" for recommended reading.

## A Virtual File System

What abstraction should be used for reading and writing to devices such as the screen and the keyboard?

A virtual file system (VFS) creates an abstraction on top of the concrete file systems. A VFS mainly supplies the path system and file hierarchy, it delegates operations on files to the underlying file systems. The original paper on VFS is succinct and well worth a read. See the section "Further Reading" for a reference.

With a VFS we could mount a special file system on the path `/dev`. This file system would handle all devices such as keyboards and the console. However, one could also take the traditional UNIX approach, with major/minor device numbers and `mknod` to create special files for devices. Which approach you think is the most appropriate is up to you, there is no right or wrong when building abstraction layers (although some abstractions turn out way more useful than others).

## 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

# System Calls

*System calls* is the way user-mode applications interact with the kernel - to ask for resources, request operations to be performed, etc. The system call API is the part of the kernel that is most exposed to the users, therefore its design requires some thought.

## Designing System Calls

It is up to us, the kernel developers, to design the system calls that application developers can use. We can draw inspiration from the POSIX standards or, if they seem like too much work, just look at the ones for Linux, and pick and choose. See the section "Further Reading" at the end of the chapter for references.

## Implementing System Calls

System calls are traditionally invoked with software interrupts. The user applications put 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 dependent on the kernel, Linux uses the number `0x80` to identify that an interrupt is intended as a system call.

When system calls are executed, the current privilege level is typically changed from PL3 to PL0 (if the application is running in user mode). To allow this, the DPL of the entry in the IDT for the system call interrupt needs to allow PL3 access.

Whenever inter-privilege level interrupts occur, the processor pushes a few important registers onto the stack - the same ones we used to enter user mode before, see figure 6-4, section 6.12.1, in the Intel manual [@intel3a]. What stack is used? The same section in [@intel3a] specifies that if an interrupt leads to code executing at a numerically lower privilege level, a stack switch occurs. The new values for the registers `ss` and `esp` is loaded from the current Task State Segment (TSS). The TSS structure is specified in figure 7-2, section 7.2.1 of the Intel manual [@intel3a].

To enable system calls we need to setup a TSS before entering user mode. Setting it up can be done in C by setting the `ss0` and `esp0` fields of a "packed struct" that represents a TSS. Before loading the "packed struct" into the processor, a TSS descriptor has to be added to the GDT. The structure of the TSS descriptor is described in section 7.2.2 in [@intel3a].

You specify the current TSS segment selector by loading it into the `tr` register with the `ltr` assembly code instruction. If the TSS segment descriptor has index 5, and thus offset `5 * 8 = 40 = 0x28`, this is the value that should be loaded into the register `tr`.

When we entered user mode before in the chapter "Entering User Mode" we disabled interrupts when executing in PL3. Since system calls are implemented using interrupts, interrupts must be enabled in user mode. Setting the IF flag bit in the `eflags` value on the stack will make `iret` enable interrupts (since the `eflags` value on the stack will be loaded into the `eflags` register by the assembly code instruction `iret`).

## Further Reading

- The Wikipedia page on POSIX, with links to the specifications: http://en.wikipedia.org/wiki/POSIX
- A list of system calls 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 [@intel3a] sections on interrupts (chapter 6) and TSS (chapter 7) are where you get all the details you need.

# Multitasking

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 notice) between the processes. At any given moment there is only one 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 the only option is to fake it. The part of the operating system responsible for rapidly switching between the processes is called the *scheduling algorithm*.

## 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 one that is specified by a path to the location of a program 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 the section "Setting up for user mode" in the chapter "User Mode".

## Cooperative Scheduling with Yielding

The easiest way to achieve rapid switching between processes is if the processes themselves are responsible for the switching. The processes run for a while and then tell the OS (via a system call) 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 on the kernel heap in a structure that represents a process. When changing to a new process all the registers must be restored from the saved values.

Scheduling can be implemented by keeping a list of which processes are running. The system call `yield` should then run the next process in the list and put 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 the `iret` assembly code instruction in exactly the same way as explained in the section "Entering user mode" in the chapter "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. Since cooperative scheduling is deterministic, it is much easier to debug than preemptive scheduling.

## Preemptive Scheduling with Interrupts

Instead of letting the processes themselves manage when 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. This kind of scheduling is called *preemptive scheduling*.

### Programmable Interval Timer

To be able to do preemptive scheduling the PIT must 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: a byte is sent to an I/O port. The command port of the PIT is `0x43`. To read about all the configuration options, see the article about the PIT on OSDev [@osdev:pit]. 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 the divider. The PIT operates at 1193182 Hz as default. Sending the divider 10 results in the PIT running at `1193182 / 10 = 119318` Hz. The divider can only be 16 bits, so it is only possible to configure the timer's 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 sent to the channel 0 data I/O port of the PIT, but since only one byte can be sent at at a time, the lowest 8 bits of the divider has to sent first, then the highest 8 bits of the divider can be sent. The channel 0 data I/O port is located at `0x40`. Again, see the article on OSDev [@osdev:pit] 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 overwrite what the previous process 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 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 occurred (the process was interrupted in user mode) 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) the CPU won't push the `esp` register on the stack. Furthermore, if there was no privilege level change, the CPU won't change stack to the one defined it the TSS.

This problem is solved by calculating 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, one must think of how to handle case when switching to a new process that should be running in kernel mode. Since `iret` is being used 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

# References