

Zeos

Yolanda Becerra, Juan José Costa and Alex Pajuelo

Course 2025–2026



PREFACE

The aim of this document is to guide the design and implementation details of a simple operating system called ZeOS.

The document will describe an initial basic implementation, to which a number of functionalities will be added. This basic implementation is responsible for booting up the OS and initialize all the necessary data structures it requires to control the hardware and start an empty unprivileged process. Both high-level (C) and low-level (assembly) programming languages will be used in order to add these new functionalities to the system.

The first piece of work to be completed will be on an essential part of any OS: the boot process. This will be followed by a section on the management of some basic interrupt and exception handling. Work will then be undertaken on process management in the OS.

For this document to be understood, the following concepts must be known:

- Mechanisms to enter the system (exceptions/interrupts/system calls).
- Process management (data structures/algorithms/scheduling policies/context switch/related system calls).
- Input/output management (devices/file descriptors).
- Subroutines and exceptions.
- Memory management.

What you should do with this document

You must first read all the documentation so that you have an overall vision of ZeOS. After that, it is advisable that you follow all the steps described in this document to design and implement some of the components of the OS such as new data structures, functions, algorithms, etc.

CONTENTS

1	PRIOR KNOWLEDGE	5
2	INTRODUCTORY SESSION	5
2.1	Working environment	5
2.2	Getting started	6
2.3	Introduction to ZeOS	6
2.3.1	ZeOS source code files	7
2.3.2	ZeOS binary image	8
2.3.3	Image construction	9
2.4	Boot process	10
2.5	Bochs	11
2.5.1	Debugging using Bochs	11
2.5.2	Frequently used debugging commands	14
2.6	Work to do	14
2.6.1	Initial steps: Understanding what is being executed	14
2.6.2	User code modification	17
2.6.3	Use of assembly	17
2.6.4	System code modification	18
3	MECHANISMS TO ENTER THE SYSTEM	20
3.1	Preliminary concepts	21
3.2	Function name conventions	21
3.3	Files	21
3.4	Hardware management of an interrupt	22
3.4.1	Task State Segment (TSS)	22
3.5	Zeos system stack	24
3.6	Programming exceptions	24
3.6.1	Writing the service routines	25
3.6.2	Exception parameters	25
3.6.3	Writing the handler	26
3.6.4	Initializing the IDT	26
3.7	Programming interrupts	27
3.7.1	The keyboard interrupt management	27
3.7.2	Writing the service routine	28

3.7.3	Writing the handler	28
3.8	Programming system calls	29
3.8.1	Independence from devices	30
3.8.2	Writing the write wrapper	30
3.8.3	Parameter passing	30
3.8.4	Returning results	31
3.8.5	Service Routine to the write system call	31
3.8.6	Copying data from/to the user address space	32
3.8.7	Writing the handler	32
3.8.8	IDT initialization	34
3.9	Programming fast system calls	34
3.9.1	Writing the wrapper	35
3.9.2	Writing the handler	35
3.9.3	Initializing fast system calls	36
3.10	Work to do	37
3.10.1	Complete Zeos Code	37
3.10.2	Implement the keyboard management	37
3.10.3	Implement the <i>write</i> system call.	37
3.10.4	Clock management	38
3.10.5	Gettime system call	39
3.10.6	Manage Page Fault exceptions	39
4	ACKNOWLEDGEMENTS	40
	Appendix	40

1 PRIOR KNOWLEDGE

For this document to be understood, it is assumed that you have acquired knowledge from other courses and that you are able to work in specific environments, specifically:

- Use a Linux environment.
- Write programs using the C language.
- Write programs using the Linux i386 assembly language.
- Add assembly code to a C file.
- Modify a Makefile to add new rules or modify existing ones.

If you lack any of these skills, you should find additional information to that given on this course. The information in the bibliography section on the course's web page may also be useful.

2 INTRODUCTORY SESSION

The main objectives of this section are:

- Become familiar with the working environment.
- Learn about the tools that must be used.
- Start analyzing and modifying the ZeOS code.
- Learn Bochs basic commands.
- Refresh some of the concepts needed.

2.1 Working environment

When developing an operating system (OS), you need a working environment usually offered by another operating system. In this case, we will develop ZeOS, a very simple operating system. To develop ZeOS we will use an Ubuntu system with the typical C and assembly toolchains and a virtual machine to test our progress. We use the following configuration:

- OS. Ubuntu 22.04¹
- GCC Compiler.
- Emulator. Bochs version 2.6.7 (<https://bochs.sourceforge.io/>)

This environment will enable you to generate the ZeOS operating system, which is a simple binary file. Once ZeOS has been generated, it can be used to boot your computer (after copying it to a floppy disk). However, you should bear in mind that the OS will be recompiled and loaded many times, so it is advisable to run your OS on an architecture emulator (like *Bochs*) in order to save time. This way, in case of problems only the emulator has to be rebooted rather than the whole computer.

Whenever you need to compress and extract your files use the following commands:

- Compress:

```
$ tar czvf file_name.tar.gz file_list_to_compress
```

- Extract:

1. The system has the user *alumne* and password *sistemas*.

```
$ tar xzvf file_name.tar.gz
```

These commands use *tar* and *gzip* at the same time.

In Ubuntu, you will be able to work with several editors (GVim, emacs, nedit, etc.).

2.2 Getting started

This section describes a quick ZeOS startup for the impatient. The following steps are performed supposing that the student works with an standard installation of Ubuntu:

- 1) Download and install ZeOS:
 - a) ZeOS is available as a compressed file (.tar.gz) at the web page².
 - b) To uncompress it, execute:

```
$ tar xzfv zeos.tar.gz
```

- 2) Enter the created directory zeos and test the OS:
 - a) Generate your ZeOS (*zeos.bin*):

```
$ make
```

- b) Execute it using bochs with internal debugger:

```
$ make emuldbg
```

- c) If everything worked well, a new window similar to the one in Figure 1 will appear (it will show the output of your ZeOS) and the prompt for the internal debugger will show the current address to be executed and wait for new debug commands. Your emulated computer is ready to boot your ZeOS image. Press *c* and *INTRO* to check that it works and *Ctrl-C* to finish. Spoiler: Nothing really amazing will happen at this point.³

2.3 Introduction to ZeOS

The construction of an OS is similar to the construction of an ordinary executable. This document will show you how an OS is built from the source code. The construction is very similar to the Linux OS building process. With minor changes, this documentation may be useful to explain how the Linux OS is built.

ZeOS is the skeleton of a simple Operating System based on a Linux 2.4 kernel, and developed for the intel 80386 architecture. ZeOS was first developed by a group of students from the Barcelona School of Informatics (FIB), with the support of a number of professors from the Department of Computer Architecture (AC). Anyone is free to add more functionalities to this OS and to make further contributions.

After downloading and uncompressing the ZeOS source code, the fastest way to build your ZeOS is to type *make* in the directory with the files:

2. <http://docencia.ac.upc.edu/FIB/grau/SO2/documents/zeos.tar.gz>

3. This will be your task, build an amazing OS (or at least one to be proud of).

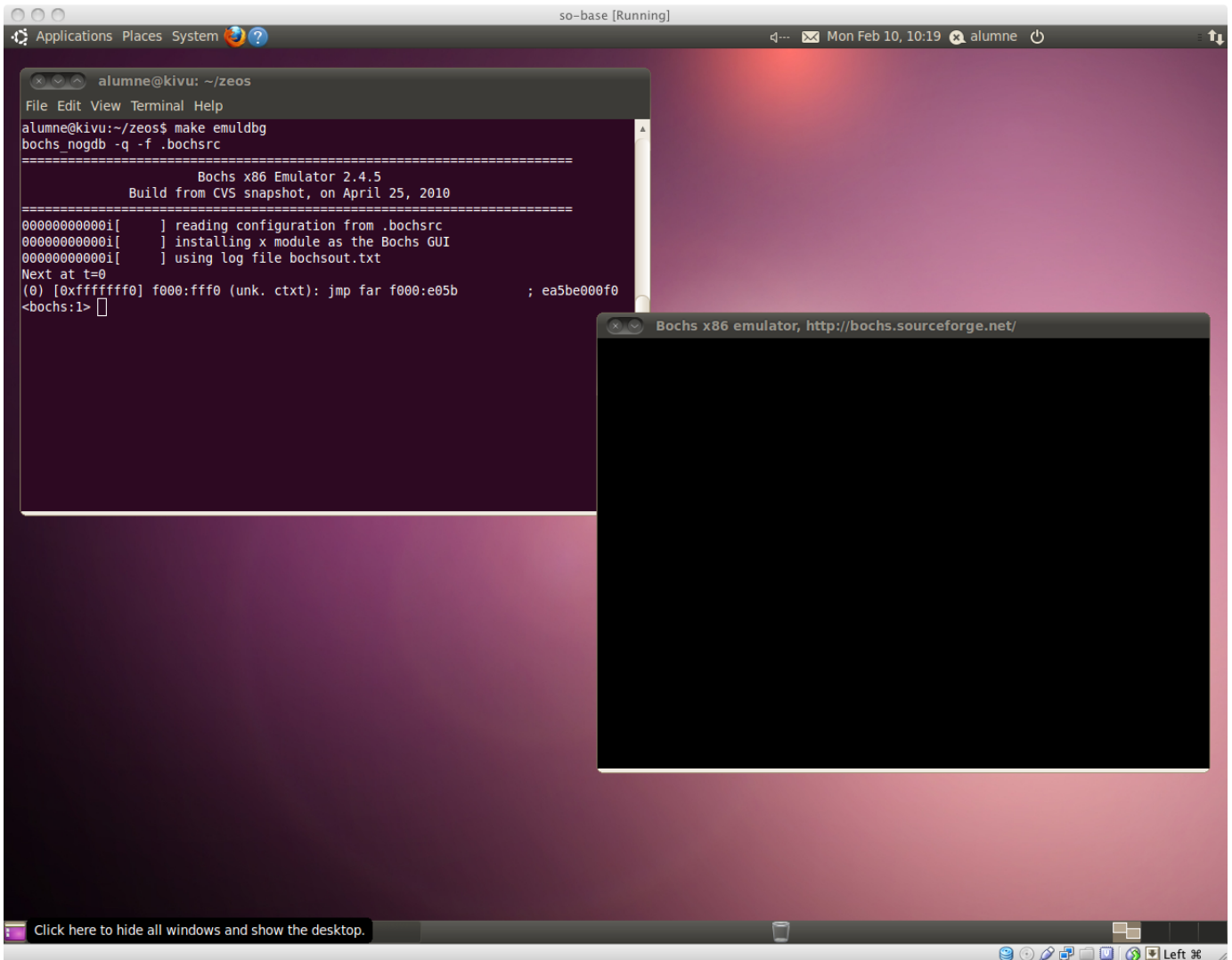


Fig. 1. Working environment with the Bochs commands window (left) and Bochs emulation window (right)

```
$ make
```

The *make* process will follow the *Makefile* rules to compile all the source files and link them together to build a final bootable image (a file called *zeos.bin*) containing your operating system. This process is explained in detail in the following subsections.

2.3.1 ZeOS source code files

The source code of your ZeOS contains files and directories. The content of the files can be divided into the following groups, depending on their extension:

- *.c*: Source files written in C language.
- *.S* (capital S): Source files written in Intel 80386 assembly language with *preprocessor* sentences. The *.c* and *.S* files should be the only ones that add code to the OS.
- *.lds*: Scripts used by the *ld* linker to combine the various files in a single binary file.

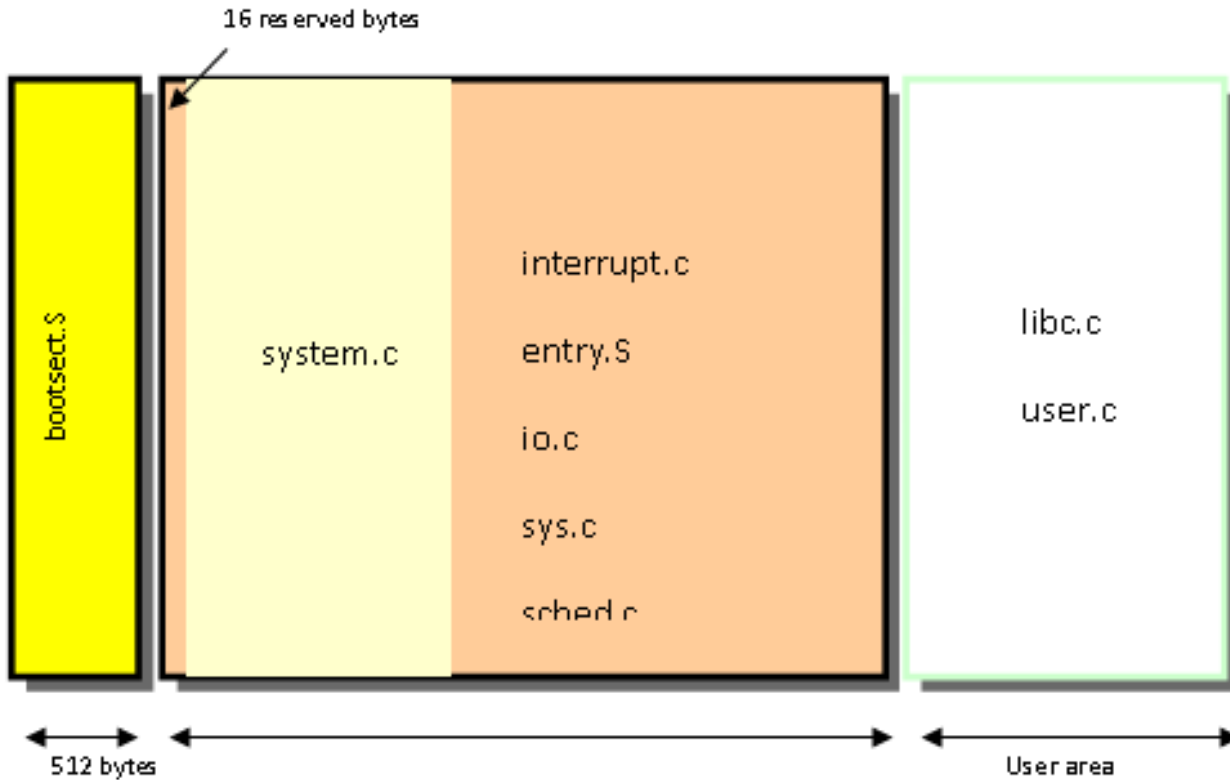


Fig. 2. The three main blocks (bootloader, system and user) forming the ZeOS binary image.⁴

- `.h`: Header files located in a dedicated *include* directory, as occurs in Linux.
- `.a`: Library files.
- (none): Only the *Makefile*, that has the necessary steps to build the OS.

Usually, there is a header file for each source file in C language, which includes all variables, functions, macros and type declarations used in this C file in case you wish to use them from another C file.

2.3.2 ZeOS binary image

After the *make* process you should obtain a single file called *zeos.bin* with the bootable image of your ZeOS.

This file is composed of three main blocks (as shown in Figure 2):

- 1) The **boot sector block**. This block corresponds to the bootloader, and it is generated by the `emphbootsect.S` file. It must weigh exactly 512 bytes, since it has to fit in a single disk sector.
- 2) The **system area block**. This block contains the resulting binary of your OS. This block will be placed in a part of the memory that guarantees its execution with the processor's maximum privilege level. It will offer services to access the hardware and others.
- 3) The **user area block**. This block has the user program. The code inside will be executed by the processor with a minimum privilege level. Therefore, it will ask the operating system (through system calls) to access any hardware or other system services that are privileged.


```

ENTRY(main)
SECTIONS
{
    . = 0x10000;

    .text.main : {
        . += 16; /* reserve space for metadata */
        *(.text.main)
    }

    .text : { *(.text) }
    .rodata : { *(.rodata) }
    .data : { *(.data) }
    .bss : { *(.bss) }

    . = ALIGN(4096); /* Magical section that marks the final size of the system data */
    .data.last : { *(.data.last) }
}

```

Fig. 3. `system.lds` file. Linker script file for the system image.

```

int __end_system __attribute__((__section__(".data.last")));

```

Fig. 4. Variable `__end_system` annotated with the section `.data.last` (in `mm.c`)

One may ask why the user program is attached to the OS when this actually never happens. Usually, a disk contains a boot loader and a file system. As we will see in next Section 2.4, the boot loader knows where the system binary resides in the file system, loads itself and then it loads the user programs, which also resides in that file system in some directory. The simple answer is that currently there is no file system nor an executable loader. Therefore, the user program is attached to the OS (in a position that is easy to calculate), and it is manually copied (*loaded*) from its location in the image to a memory area with user privileges to which execution is transferred.

Figure 3 shows the contents of the `system.lds` file that enables the linker know how to locate in the memory the various sections of an executable file. In this case, first, it sets the entry point to the binary (the initial function to be called on start); second sets the starting address (0x10000) of the system code; reserves some space for metadata, by increasing the current address by 16 bytes, that we will use to store the system image size and the user image size (with some extra bytes reserved for future uses); then the system code (`.text`); the read-only data section (`.rodata`); the initialized data (`.data`); the non-initialized data (`.bss`); and, finally, it leaves a gap to align the current address to a page-size (4096 bytes) address and adds a special section (`.data.last`) which will contain the `__end_system` variable defined in `mm.c` file (shown at Figure 4) that marks the end of the system image (will be the last used address).

2.3.3 Image construction

The way the OS image is built will be explained backwards. This process is done automatically using the *Makefile* provided with ZeOS. First, we will assume that the three blocks (*boot* sector,

4. Some of the file names that appear in Figure 2 may not match the ones from your currently downloaded source code.

system area and *user* area) have been generated and then we will see how to put them together into a single file like the snapshot shown in Figure 2.

The program that attaches the files is called *build* (*build.c*). This program will receive three binary files, and will output in the standard output a single consecutive file with the required metadata (system and user size). This program is generated as follows:

```
$ gcc -Wall -Wstrict-prototypes -o build build.c
```

To attach the three blocks one after the other, it is only necessary to execute the following command:

```
$ ./build bootsect system.out user.out >zeos.bin
Boot sector    512 Bytes.
System        16388 Bytes ( 16 kB)
User          82206 Bytes ( 80 kB)
Disk Image    99106 Bytes ( 96 kB)
```

Where *bootsect* is the binary content of the boot sector; *system.out* is the binary content of the system area (just code and data); *user.out* is the binary content of the user area; and *zeos.bin* is the resulting binary of the OS. *build* checks block sizes, adds the user and system sizes and writes this value in a specific boot sector position, specifically in bytes 500 and 501, which are labeled in the *bootsect.S* as *sysize*. This will be used by the bootloader to load the whole system (and user) image in memory. Once in memory, the OS code must move the user part to its final position, the user code entry point address, so it needs the total user size. That is why 16 bytes are reserved at the beginning of the system block, as can be seen in Figure 2. They are initially empty but the *build* program writes them with the sizes of the system and user images.

2.4 Boot process

The blocks *boot*, *system* and *user* (described in the previous section) are appended to each other to create a single binary format file with the content of all ZeOS operating system. This file may be copied to an unformatted floppy disk⁵, beginning at sector 0, and any computer booting this floppy disk would start our ZeOS automatically.

The boot process is simple. After turning on your computer (before the OS is executed) a program is placed in a read only memory (ROM) area and it will be executed, called the BIOS. This program checks that the PC is working properly.⁶ It then looks for the preconfigured booting device and tries to access it. It does so by loading the initial sector from this device (sector 0) into the memory. It does not matter whether the device is a floppy disk, a hard disk, or a CD-ROM.

As the size of the first sector that the BIOS loads is quite small (512 bytes), there is not enough space to store the entire ZeOS. It is only possible to store a loader (the *bootloader*). This bootloader is designed to fit into these 512 bytes and the BIOS just loads them into the memory at a hard coded address (0x7C00). Once these 512 bytes are copied to the memory, the BIOS transfers the execution to the first byte of this small block. The bootloader starts its execution at this point.

The main task of this bootloader is to locate the operating system binary image, load it into the memory and transfer the execution to it. In our case, the binary image is located after the boot

5. For historical reasons... just look here https://en.wikipedia.org/wiki/Floppy_disk :)

6. You can see how the memory, the disk and other devices are checked during this process.

sector (the system and user areas), and its size is hard coded into the bootloader itself. Therefore the bootloader finishes loading what is left on the floppy disk (system and user blocks) at the memory address 0x10000 and, finally, it transfers the execution to the first byte of the system code (corresponding to the *main* routine from system.c) starting the execution of your ZeOS.

The ZeOS starting code initializes itself and prepares to run the user code. The user code is loaded at a wrong address, therefore it is required to move it to an unprivileged memory area (the user area that starts at 0x100000). Finally the system initialization code transfers the execution to this user area, starting the execution of the unprivileged code.

2.5 Bochs

Bochs is a PC Intel x86 emulator written in C++. It was created in around 1994. Originally it was not free, but when Mandrake bought it, it was granted a GNU LGPL license. It is a little slow, although not to any noticeable extent for the purposes of this project. However, it is very reliable. In this document we will use **version 2.6.7**.

The Bochs executable is `/usr/local/bin/bochs`. This will read a configuration file to prepare the emulated computer and will start its execution.

Bochs configuration file

Bochs uses the `.bochsrc` file to configure the features of the emulated computer⁷. We will focus on the following three features:

- **Image location.** Line 5 defines a floppy disk drive with the image file to be inserted. The image file must correspond to the path that points to the `zeos.bin`. Currently it is set to load the image `zeos.bin` from the current directory.

```
floppya: 1_44= ./zeos.bin, status=inserted
```

- **Boot device.** Line 6 defines the device used to boot the emulated computer. In our case, the floppy device.

```
boot: floppy
```

By default, Bochs assumes a default configuration file named `.bochsrc`. But, you can use the `-f` option in order to use another configuration file⁸:

```
$ bochs -f .bochsrc_gdb -q
```

2.5.1 Debugging using Bochs

In order to debug your operating system, it is necessary to control execution using a debugger. Bochs offers two options for debugging code: (1) using GDB as an external debugger or (2) using an internal debugger that is part of the emulator. Both options are *exclusive*, and therefore there are

7. The details on the different options used may be find at <https://bochs.sourceforge.io/doc/docbook/user/bochsrc.html>

8. The target `gdb` from the Makefile uses this option.

two different executable files available in the laboratory (*bochs* and *bochs_nogdb*). You can choose the best version taking into account that:

- GDB is more suitable to debug high-level programming problems (not related with the underlying hardware).
- The internal debugger is recommended when you want to debug specific information about the emulated hardware (namely physical addresses translation, special registers, ...).

In summary, use GDB version (*bochs*) by default and only use the Bochs internal debugger (*bochs_nogdb*) when everything else fails.

Controlling execution through an external GDB debugger

In order to start the GDB debugger you can use:

```
$ make gdb
```

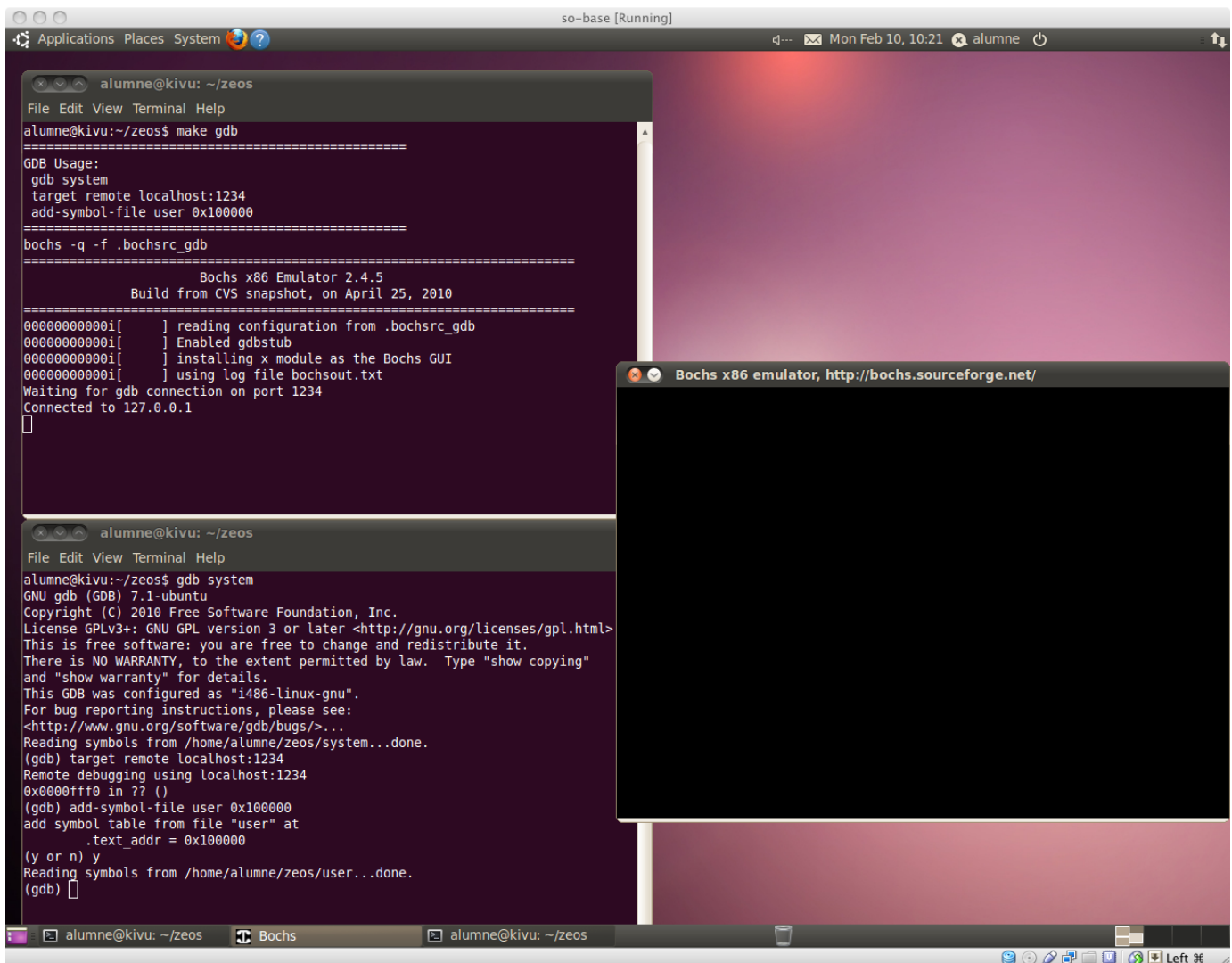


Fig. 5. Bochs commands windows (up), Bochs emulation window (right) and GDB window (down) after establishing a connection between GDB and Bochs.

Figure 5 shows the windows appearing with this command: the emulation window, the Bochs window and the GDB window with its prompt⁹ ready to accept commands (add breakpoints, continue the program execution, etc). You can find some help on debug commands in section 2.5.2 or the GDB reference guide with all the GDB commands on the following link: <https://sourceware.org/gdb/download/onlinedocs/>

Under the hood the previous command follows the steps below:

- Compile the ZeOS code with debug symbols, using GCC with the "-g" flag (which is enabled by default in the Makefile).
- Execute Bochs with support for the external debugger and load the configuration file that enables debugging with GDB (*.bochsrc_gdb*):

```
$ bochs -f .bochsrc_gdb -q
```

- In this file the option for *External debugger configuration* is added. When Bochs is compiled with GDB support, it starts the emulation with debug disabled. In order to use GDB it is necessary to add the following line to the configuration file:

```
gdbstub: enabled=1, port=1234, text_base=0, data_base=0, bss_base=0
```

This option instructs Bochs to start the emulation and wait for a connection at port 1234 from a GDB session to control the execution. It should be highlighted that this line is only interpreted when the Bochs is compiled to use the external debugger. Otherwise, it will complain with an error about the lack of external debugger support.

The Bochs virtual machine will start (opening the emulation window) and will wait for the connection request from GDB.

- After executing Bochs, a new terminal is needed to execute GDB.¹⁰ GDB accepts a program filename as the argument to be debugged, and it automatically loads all the symbols of this program. Therefore, start GDB to debug the system part of our Zeos:

```
$ gdb system
```

- Connect GDB to the Bochs instance by executing the command below in GDB (the port should be the same as the one used in *.bochsrc_gdb*).

```
(gdb) target remote localhost:1234
```

- And, finally, load the remaining symbols for the user part.

```
(gdb) add-symbol-file user
```

Controlling execution through internal Bochs debugger

To start the Bochs internal debugger you can use:

```
$ make emuldbg
```

9. At the GDB prompt you may want to switch to the gdbTUI using the combination 'C-x a'. Look for more information at <https://sourceware.org/gdb/onlinedocs/gdb/TUI.html>

10. Instead of GDB you can use any front-end to GDB, like DDD or gdbTUI.

Once executed, you should see two windows like the ones shown in Figure 1: the emulation window (where the output to the screen device will be shown) and the Bochs window with a prompt (where debug commands may be inserted).

Under the hood the previous command is following the next steps:

- Compile the ZeOS code with debug symbols, using GCC with the "-g" flag (which is enabled by default in the Makefile).
- Execute Bochs with internal debugger enabled using the default configuration file (*.bochsrc*).

```
$ bochs_nogdb -q
```

The commands window displays a prompt (*<bochs:1>*) in which commands can be introduced. This prompt shows information about the memory address that is about to execute (corresponding to the register EIP content, *'0xffffffff0'* in this case) and its memory content (showing also its translation to an assembly instruction, *'jmp far f000:e05b'* in this case). A summary of the commands that can be executed may be found at <https://bochs.sourceforge.io/doc/docbook/user/internal-debugger.html>.

2.5.2 Frequently used debugging commands

The gdb manual page (*man gdb*) is a good starting point for someone new to debugging applications, explains how to use it to debug a program and a list of the most frequently used commands. Table 1 presents, for reference, some of the most frequently used commands for Bochs and GDB debuggers. Also you could find online tutorials on GDB.¹¹

2.6 Work to do

In order to familiarize with the environment is advisable to carefully follow and try the steps described in this section. You will learn how to:

- Generate the ZeOS image.
- Visualize the generated object code (user and system).
- Relate variables and functions from code to its location in memory (and the other way around).
- Use the debugger to control the execution of your code and view information about data and some hardware content (memory and registers).
- Modify the user code.
- Call assembly functions from C routines in your code.
- Modify the system code.

2.6.1 Initial steps: Understanding what is being executed

In this section you will prove that after invoking Bochs to run the ZeOS image, ZeOS is being loaded in memory, initialized and it starts the execution of the user process which executes an infinite loop. To do that you must:

- 1) Generate the ZeOS image.

11. Like Beej's Quick Guide to GDB at <http://beej.us/guide/bggdb/> showing a sample GDB session (really useful).

12. You can find an example of stack frames at https://sourceware.org/gdb/download/onlinedocs/stack_frame.pdf.gz

Bochs	GDB	Description
help	help	Show help about the commands that can be executed. Usually this accept a command to show more specific help.
Control flow		
continue	continue	Execute the image normally until the next breakpoint (if you have one). While executing the image, you can not issue new debugger commands, so in order to stop the execution you should press the Ctrl-C key combination, and the debugger prompt will appear again.
step [num]	stepi si	Execute a single assembly instruction. If <i>num</i> is provided, execute this number of instructions.
(not available)	step	Execute a single high level instruction (C).
next	nexti ni	Execute a single assembly instruction stepping over subroutines. If the instruction is a <i>call</i> to a subroutine, the whole subroutine is executed (instead of simply executing the instruction) and execution stops at the instruction after the <i>call</i> instruction.
(not available)	next	Execute a single high level instruction (C) stepping over subroutines.
lbreak address	b *address b function	Insert a breakpoint in the instruction indicated in the address. The address can be written in decimal (123459), octal (0123456) or, as is usual, in hexadecimal (0x123abc). For example, if you write "b 0x100000" (or "b *0x100000" in GDB), it will insert a breakpoint in the first code line of the user process. GDB also accepts a function name or a specific location in a file (<i>filename:linenumber</i>).
break address	(not available)	Insert a breakpoint at a physical address.
Data examination		
r	info r	Show the content of the hardware registers .
x addr	x addr	Examine memory content at address <i>addr</i> .
print-stack [num_words]	x/16 \$esp	Print <i>num_words</i> from the top of the stack . By default, only 16 values are shown. It is only reliable if you are in system mode, when the base address of the stack segment is 0.
(not available)	print expr	Display the value of an expression or C variable.
(not available)	backtrace bt	Backtrace: Display the program stack frame ¹² for each active subroutine.
(not available)	frame number	Select a specific frame number.
info tab	(not available)	Show current address translation (Logical -> Physical).
quit	quit	Exit debugger.

TABLE 1
Frequently used commands for Bochs and GDB.

- 2) Examine the generated output of the *make* command. Look at the different commands used and their flags (the *man* command may be useful to understand their meaning) and try to relate the generated files with the different areas shown in Figure 2.
- 3) Run the generated image. For that, you will start the bochs debugger.¹³
- 4) ZeOS should start, initialize itself, and start an user process. After the message stating "Entering user mode" it seems that ZeOS hangs or it is doing nothing. Therefore, where is ZeOS hanged?, which assembly instruction is executing?
- 5) Where is the previous instruction located in the code?
 - To interpret the code execution, we need to locate this memory address into the code section of the generated image. For that, the commands *objdump* and *nm* can be used¹⁴.
 - The *objdump* command shows the compiler-generated assembly code of an object file, it also shows each instruction encoding, and its corresponding memory addresses, which will be useful to insert breakpoints later. Figure 6 shows the result of using *objdump* with the *system* file, where the content of the C function *main* is shown starting at memory address 0x10004 (you may have different addresses in your binary).

13. Remember, as stated in Table 1, that *continue* starts the execution and *Ctrl-C* stops it.

14. Look at their manuals for extra documentation, especially you should look for *-d*, *-h* and *-S* flags in *objdump* and *-n* flag in *nm*.

```

00010000 <main-0x4>:
    10000:      10 00                adc    %al, (%eax)
    ...
00010004 <main>:
    10004:      55                push   %ebp
    10005:      89 e5            mov    %esp, %ebp
    10007:      83 ec 08        sub    $0x8, %esp
    1000a:      83 e4 f0        and    $0xffffffff0, %esp
    1000d:      6a 00            push   $0x0
    1000f:      9d                popf
    10010:      ba 18 00 00 00    mov    $0x18, %edx
    10015:      b8 18 00 00 00    mov    $0x18, %eax
    1001a:      fc                cld
    1001b:      8e da            mov    %edx, %ds
    1001d:      8e c2            mov    %edx, %es
    1001f:      8e e2            mov    %edx, %fs
    ...

```

Fig. 6. "objdump -d system" output sample

```

0001042c T set_ss_pag
00010224 T set_task_reg
000102b8 T setTrapHandler
000100f8 T setTSS
00011000 D task
0001c000 D taula_pagusr
0001d020 B tss
00010538 D usr_main
0001d8a0 B x
00010544 D y
    ...

```

Fig. 7. "nm system" output sample

- The *nm* command shows the symbols (variables and functions names) present in an object file and their memory locations. For instance, Figure 7 shows that, for example, there is an initialized variable (D) *task* located at address 0x11000.

2.6.2 User code modification

At this point we are ready to modify the user part with some code. Currently the OS does not offer any service¹⁵, therefore the user application is very simple. The *user.c* file is the place in which we will write the user code. **We strongly recommend that you only modify this file for now.**

- Add to the *user.c* file a new function *add* that returns the sum of its two parameters. This function has the following header:

```
int add(int par1, int par2);
```

- Modify the *main* function in the *user.c* file to add a call to the previous function with some dummy values (like 66 and 1638) and store the resulting value in a local variable.
- Generate a new ZeOS image. Correct any errors that may appear and check for any warnings when it is compiled (a new warning must appear!).¹⁶
- Execute the command "*objdump -d user.o*" to see the generated assembly code for the file *user.c*. Note that the addresses begin at zero as it is an object file.
 - Can you see the generated code of the function *add*?
- Now look at the resulting executable file, execute "*objdump -d user*". Do the addresses match the previous object file? Why?
- Run the whole program in the bochs debugger (*continue* command). To regain control in the debugger you have to type *Ctrl+C*. After doing so, *which line is going to be executed?*
- As there is no output feedback, we will re-execute the program with breakpoints. To do this, at the debugger prompt set a BREAKPOINT at the first instruction of the *main* routine and another one at the *add* routine. Now that they are set, we could continue the execution using the *continue* command, but these breakpoints are already passed, and therefore we need to restart the bochs emulator. At the upper right zone of the bochs Window there is a button named **Reset** to do exactly this. Check that the execution stops at the first breakpoint. Then, continue the code execution and try to arrive to the 2nd breakpoint.
- After arriving to the *add* routine breakpoint, examine the memory content, the stack or the registers. *Where are the function parameters?*
- Re-execute a part of the program step by step. To do this, use the commands *step* and/or *next* to move the execution forward¹⁷

Note: Adding a BREAKPOINT is often the only useful way to reach to a subroutine address. For example, after entering a loop with a high number of iterations, and you want to get out of it. In this case, the number of STEP commands that may be needed to get out of the routine would be extremely high.

2.6.3 Use of assembly

Let's add new files to our user application using some assembly.

- Examine the generated assembly code for the *add* function and its invocation.

15. This means that you can NOT use any of the *libc* functionalities (like *printf*), but you will add these features soon.

16. The warnings do not stop the compilation, but they are a sign that something weird happens and it needs extra care. An ideal compilation phase would show no warnings, and therefore any change in the code that produces a warning will be easily spotted –otherwise it may be hidden in a ton of already ignored warnings–. After this session try to remove them all.

17. It has been seen that after stopping at a breakpoint with GDB and Bochs v2.6.7, it is unable to continue the execution with the debugger instructions *step* or *next*. Therefore you must use the commands *stepi*, *nexti* or *continue*.

```
#include <asm.h>

ENTRY (addASM)
    /* your assembly instructions separated by newlines (\n) */
```

Fig. 8. Example showing how to define the assembly function 'addASM' with the *ENTRY* macro.

- How is the dynamic link generated?
- How are the parameters accessed?
- How is the result returned?
- Create a new function *addAsm* in assembly with the same goal: receive two parameters, add them and return the resulting value (the important thing is that it must follow the C convention to be called from a C program):
 - Create a new file for user assembly code: *suma.S*. (Note the capital letter 'S', this will use the C preprocessor at compile time).
 - Include the header file 'asm.h'.
 - Define the function *addAsm* using the *ENTRY* macro¹⁸ (defined in *include/asm.h*). This macro helps to define a function header as shown in Figure 8.
 - Add your function code in assembly just below the previous header.
 - Modify Makefile to:
 - * preprocess the file and generate a *suma.s* file (similar to *entry.s* target).
 - * assemble this file and add it to the list of files to be linked for the user image (similar to *libc.o* target).
- Add a call to the new assembly function *addAsm* in the *main* function of the *user.c* file that adds the result of the previous call to *add* plus the value 51914, storing the result in a new global variable. As the function is in a different file, we must instruct the C compiler to recognize the assembly function by adding a header for it. This header gives the compiler all the information it needs (number and type of the parameters, and the resulting type) and the linker will finish the work linking the function call to its real code.
- Check with the debugger that this assembly version works correctly. Specifically, check how the parameters are passed to the different functions.
- Which address did you enter in the Bochs debugger to access the variable that stores the resulting value from the *add* function?

2.6.4 System code modification

The previous sections have shown how to modify the user application, let's look at the operating system code.

- The entry point of your operating system is the *main* routine in the *system.c* file. Skim its content and observe that there are a couple of calls to a *printk* routine that prints a message in the screen.
- Add some message you would like to be printed at the beginning of your OS.
- Find the *printk* code and try to understand what is it doing. The screen device is a memory mapped device.¹⁹ In particular, the text mode offered by this device consists on a matrix

18. You can check the bibliography for preprocessor macros documentation.

19. A better explanation about the screen device may be found online, for example at Chapter 23 of *The Art of Assembly* from Randall Hyde (<https://www.plantation-productions.com/Webster/www.artofasm.com/DOS/pdf/ch23.pdf>).

of 25 rows by 80 columns located at a specific memory address (*0xb8000*). Each cell of this matrix contains two bytes, the character and its color attributes. As the operating system is executing in a privileged level, it can access the required memory addresses to access the screen hardware device (in this case a memory mapped device).

- To understand the different privilege levels, try to replicate the *printc*, *printk* and any needed variables into the user code and make a call to *printk* from the *main* routine to show a "Hello world" message in the user code. (Spoiler: it will NOT work, but you will learn what happens when trying to access directly a hardware device²⁰ from user mode.)
- You may remove the previous modifications as they will not be needed anymore.
- As an OPTIONAL challenge, add a new *printc_color* routine, similar to *printc*, that changes the foreground color of the printed character.
- As an OPTIONAL challenge, try to modify the system *printc* function to scroll the screen content if the cursor surpasses the maximum coordinates instead of the current implementation that starts again at the first row.

20. Technically speaking it is not a 'hardware device' but a memory address with permissions for privileged accesses only.

3 MECHANISMS TO ENTER THE SYSTEM

Once the operating system boots, it gets control of all the hardware, and transfers the execution to the user application lowering its execution level to a non-privileged one. Therefore, in order to execute any privileged code we need a special mechanism to invoke it: *the mechanisms to enter the system*.

We will use the *interrupt mechanism* to execute code from the OS again. Interrupts are events that break the sequential execution of a program and they call a specific handler to solve the situation. These interrupts can be classified as synchronous or asynchronous depending on who generates them:

- Synchronous interrupts are generated by the CPU, at the end of the execution of an instruction.
- Asynchronous interrupts are generated by other hardware devices.

From now on, we will refer to synchronous interrupts as **exceptions** and to asynchronous interrupts as **interrupts**, although both follow the same mechanism.

Intel hardware provides the Interrupt Descriptor Table (IDT) to associate each interrupt with its specific function code. Each interrupt — identified by a number between 0 and 255 — has an entry in this table. Among other data, it contains the address of the routine to be executed (the **handler**) and the minimum privilege level needed to execute it.

The table, depending on the interrupt number, is divided in 3 parts: as follows:

- 0-31 are exceptions and unmasked interrupts.
- 32-47 are masked interrupts.
- 48-255 are software interrupts. These will be used by our OS to offer services to the user application. For example: Linux uses interrupt 128 (0x80) to implement the system calls.

This information is extracted from the Intel manuals. For more information, see chapter 4 of *Understanding the Linux Kernel*.

This is why we will implement three different mechanisms: 1) Exceptions, 2) Interrupts and 3) System calls or traps (syscalls).

Therefore, in order to handle a specific exception or interrupt, we will modify the IDT with a specific code. In our case we will separate this code in 2 different routines: the hardware management (the handler) and the service management (the service routine). Then, to program an exception or interrupt behavior, you must:

- 1) **Write the service routine** for the interrupt or the exception, which will be the code to service the interrupt or exception.
- 2) **Write the handler** to be executed when the interrupt or the exception is generated. The handler is the assembly code to save the hardware context of the current execution flow, call the service routine and restore the previous hardware context.
- 3) **Initialize the IDT entry** to link the exception or interrupt id number with its associated handler.

The OS offers different services to the user applications through the **system calls**. These services are privileged functions inside the OS and therefore implemented through the interrupt mechanism. In order to ease the user application code, we will use functions, named **wrappers**, to isolate low-level and non-portable code. These functions will wrap the passing of the system call parameters, the generation of the interrupt and its result processing for each system call.

The user code invokes any of these wrappers — as any other user function call — causing the actual entrance into the system, the execution of the system service, and obtaining its result back.

In this section you will find:

- Hardware management of an interrupt.
- Code to manage exceptions.
- Code to manage the keyboard interrupt.
- Code to implement system calls.
- Code to implement the write system call.

The following sections include a how-to guide to add an exception, an interrupt and a system call to your OS. Remember that **these mechanisms** — even they are described separately in this guide — **are almost the same, since they are all managed by the same interrupt mechanism (the IDT).**

3.1 Preliminary concepts

- Interruption, exception and system call.
- Context of a process.
- Hardware management of an interrupt.
- Checking parameters and returning results in a function.

3.2 Function name conventions

The functions will be given similar names to make the code easier to follow. The handler for an interrupt will be given the same name as the interrupt followed by *_handler* and the routine services will be given the same name as the interrupt or exception, followed by *_routine*. For example, the handler name of the exception *divide error* will be *divide_error_handler* and its routine service name will be named *divide_error_routine*.

As regards the system calls, the handler will be named *system_call* and the service routines will be *sys_namesyscall*, where *namesyscall* corresponds to the specific system call name (for example *sys_write*).

3.3 Files

ZeOS files generally have functions that share a common objective or type. The main files (probably not the only ones) in this section are:

- *system.c*: System initialization (main)
- *entry.S*: Entry points to the system (handlers)
- *interrupt.c*: Interruption service routines and exceptions
- *sys.c*: System calls
- *device.c*: Device dependent part of the system calls
- *wrappers.S*: System call wrappers
- *libc.h*: System call headers
- *libc.c*: Other user level code provided by the OS (*perror*, *errno*, ...)
- *io.c*: Basic management of input/output

3.4 Hardware management of an interrupt

Once the system has been initialized, the hardware **automatically** performs the following steps when an interrupt has been generated (for more information, see *Understanding the Linux Kernel*):

- Determine the i index of the interrupt vector and access the corresponding IDT entry.
- Verify that the interrupt has enough privileges to execute the handler, by comparing the current privilege level with the one stored in the IDT entry. If access is unauthorized it generates a general protection exception.
- Check the privilege level of the handler routine to see if it is different from the current execution level — our case — in which case the stack will have to be changed.
- Change from the current user stack to the system stack, the hardware obtains the address of the system stack to be used from the Task State Segment (TSS) through the SS and the ESP0 fields. Then the hardware saves in this new stack: 1) the current content of the SS and ESP registers (they point to the top of the current user stack), 2) the value of the EFLAGS (the cpu state word), and 3) the contents of CS and EIP registers (they contain the address of the next instruction to be executed in user code). The resulting stack is shown in Figure 9.
- Execute the code at the address saved in the i^{th} entry of the IDT.

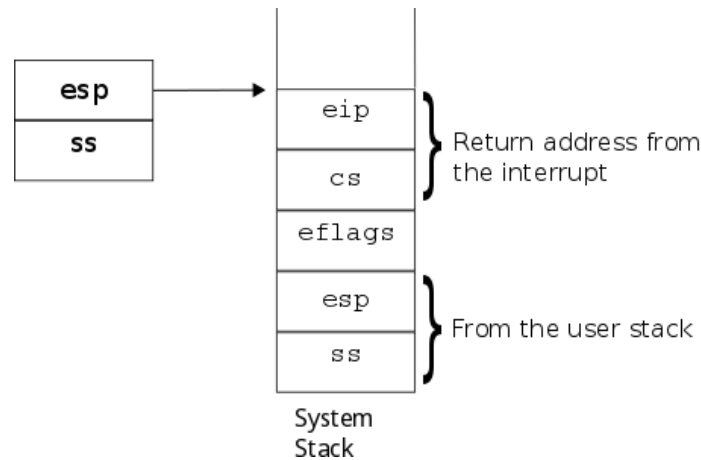


Fig. 9. State of the system stack when the interrupt handler begins execution

Once the interrupt management code completes, the control returns to the instruction that generated the interrupt by executing the `iret` instruction. The `iret` instruction takes the new values of the EIP and CS registers from the top of the stack, loads the EFLAGS registers with the stored value in the stack and modifies the ESP and SS registers to point to the stack that was used before the interrupt happened (user stack in our case).

3.4.1 Task State Segment (TSS)

The 80x86 architecture defines a specific segment called the task state segment (TSS) to store task related information in memory. It is mainly used to implement a context switch in hardware. Neither ZeOS nor Linux want to use the TSS but the architecture forces them to define a TSS for each cpu (1 in our case). **We use the TSS to know the address of the system stack when making a user mode to system mode switch.** You can check the ZeOS files to see the fields that the TSS has and how it is initialized for the initial process.

Figure 10 shows how the `ss0` and `esp0` fields of the TSS are used when switching from user mode to system mode. These fields always point to the bottom of the system stack.

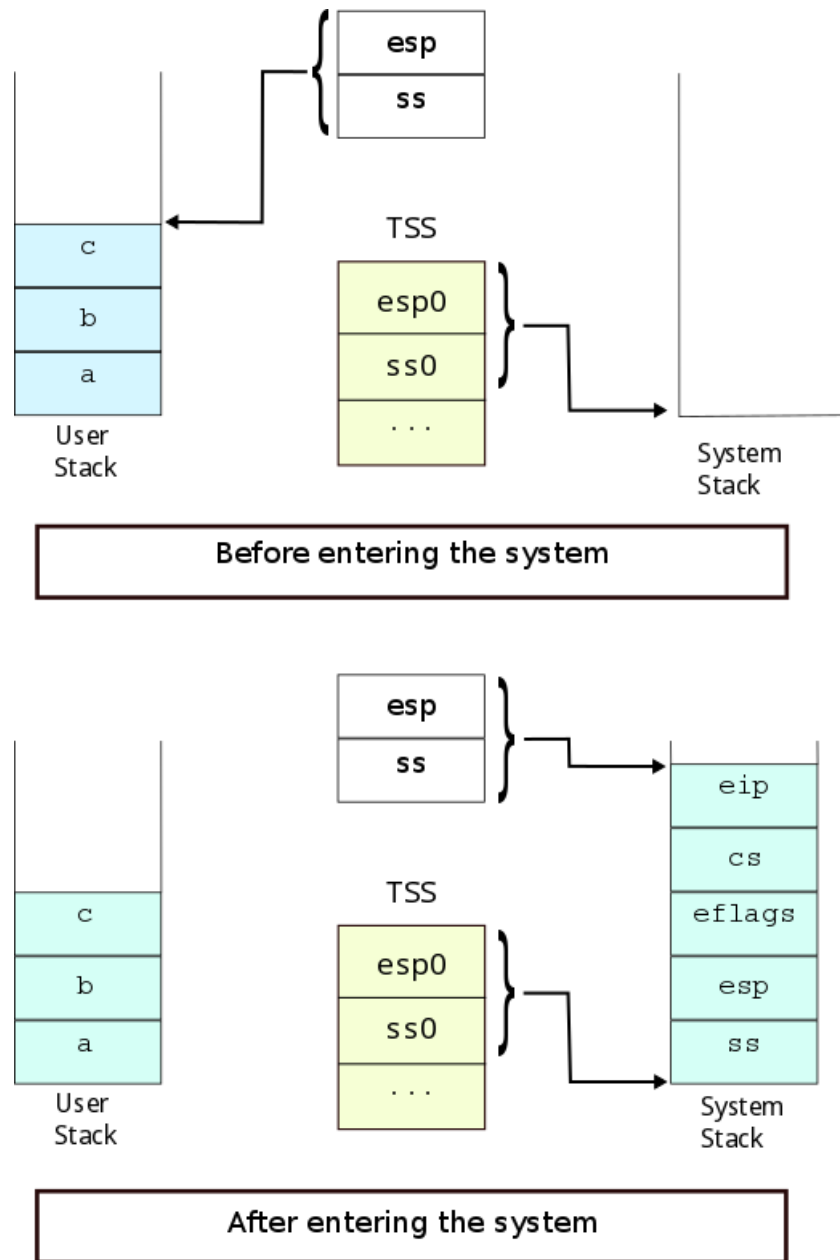


Fig. 10. The esp0 and ss0 fields of the TSS show the location of the system stack used in switch from user mode to system mode

3.5 Zeos system stack

High level operating system routines (typically `sys_*`) expect the same contents of the system stack when they start execution. Therefore the Zeos system stack must always be constructed as shown in Figure 11.

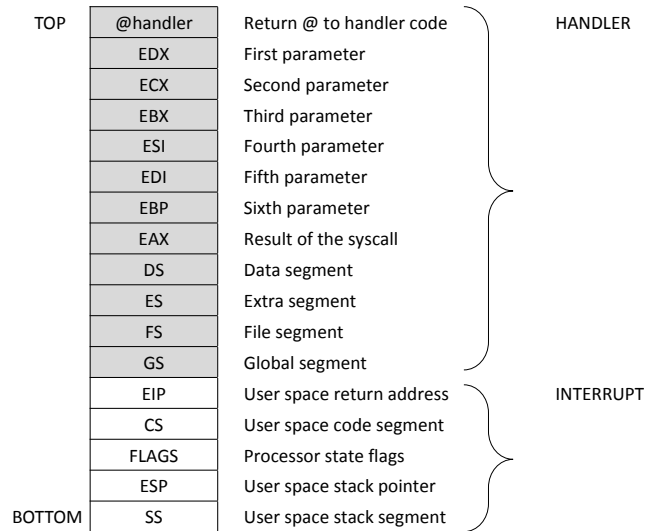


Fig. 11. Zeos system stack content at any system service routine after being called by the handler

From bottom to top, the stack begins with the minimal information needed by the processor to return to user mode (white part of the stack in Figure 11). This information is pushed by the processor when executes an `INT` instruction to switch to system privileged level (see Section 3.4) and consists on the address of the top of the user stack (registers `SS` and `ESP`), the processor's state and the return address to the user mode code (registers `CS` and `EIP`).

In top of that, the operating system programmer must decide what information is important in system mode to be restored when switching back to user mode (grey part of the stack in Figure 11). All this information is pushed in the stack inside the system call handler. In Zeos, this information is composed by the segment registers (`GS`, `FS`, `ES` and `DS`), the kernel return value, and the parameters needed by the service routine, pushed from right to left. Finally, since the handler performs a call to the service routine, the return address to the handler is also pushed in the stack.

It is very important that, to ensure the full compatibility of the operating system routines with the system calls (see Section 3.8) and fast system calls (see Section 3.9) mechanisms, the OS dependent part of the stack pushed in the handler has to be the same, and needs to be always in the same offset of the stack. If this requirement is not accomplished, the operating system will not be able to work with syscalls and fast syscalls at the same time.

3.6 Programming exceptions

We will program system exceptions in this section. Whenever an exception is produced the normal execution flow is interrupted and the OS gets the control through the IDT. Table 2 shows the exception positions in the IDT, their names and the number of bytes of their parameters (or error

code²¹) if they exist. A real OS tries to recover from the exception and continue the usual execution, but in ZeOS the exception service routines will be very simple: they will show a message in the screen with the exception that has been generated and they will enter in an infinite loop state (never returning to the user mode and therefore hanging the system).

# IDT	Exception	Parameter
0	Divide error	No
1	Debug	No
2	NM1	No
3	Breakpoint	No
4	Overflow	No
5	Bounds check	No
6	Invalid opcode	No
7	Device not available	No
8	Double fault	4 bytes
9	Coprocessor segment overrun	No
10	Invalid TSS	4 bytes
11	Segment not present	4 bytes
12	Stack exception	4 bytes
13	General protection	4 bytes
14	Page fault	4 bytes
15	Intel reserved	No
16	Floating point error	No
17	Alignment check	4 bytes

TABLE 2
List of system exceptions

3.6.1 Writing the service routines

The exception management in this operating system will be very simple. Whenever an exception is raised, the OS will show a message with a short description and will wait in an infinite loop, stopping the system.

For example, the code of the general protection service routine is:

```
void general_protection_routine()
{
    printk("\nGeneral protection fault\n");
    while(1);
}
```

All exception service routines are already implemented inside the *libzeos.a* library.

3.6.2 Exception parameters

As stated in Table 2 some exceptions receive additional information from the hardware in order to be solved (for instance, information on the type of access that generates the page fault exception). This information has a fixed size (4 bytes) and is pushed into the system stack automatically. Figure 12 shows the system stack contents and the location pointed by the *esp* and *ss* registers when an exception with an error parameter is raised.

21. For some exceptions, the CPU generates a hardware error code and puts it in the system stack before it starts the execution of the exception handler.

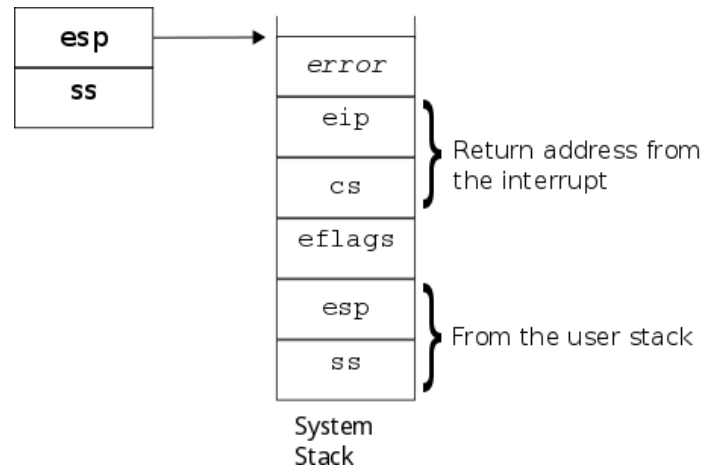


Fig. 12. System stack contents when an exception with an error parameter is raised

3.6.3 Writing the handler

A handler function must be written for each exception. All exception handlers have a common scheme and must follow these steps (in assembly):

- 1) Define the function header in assembly. Use the **ENTRY** macro²² with the name of the exception handler as a parameter (*ENTRY(handler_name)*) to do it.
 - From this point on, the **ENTRY** macro will always be used in assembly to define a function header.
- 2) Save the context in the stack (use the **SAVE_ALL** macro).
- 3) Call the service routine.
- 4) Restore the context from the stack (use the **RESTORE_ALL** macro). Notice that the order of restoring the values of the registers from the stack must match the order in the **SAVE_ALL** macro.
- 5) If needed, clear the error code removing it from the stack. Check Table 2.
- 6) Return from the exception function. Since it is not a "normal" function return because it has to change mode, an **iret** rather than a **ret**²³ instruction must be used.

All exception handlers are already implemented inside the *libzeos.a* library.

3.6.4 Initializing the IDT

Finally, to initialize a position in the IDT, the following function is provided:

```
setInterruptHandler(int position, void (*handler)(), int privLevel)
```

The parameters required to write the exception are:

- `int position`. Entry in the IDT to modify.
- `void (*handler)()`. Address of the handler that will handle the exception²⁴.

22. Check the preprocessor document from the references to know how the macro mechanism works.

23. You need to know which data are saved in the stack when going into system mode and what the *iret* instruction does.

24. This is a *function pointer*, you may consult the K&R book "The C Programming language" to refresh your knowledge about them.

- `int privLevel`. Privilege level needed to allow the execution of the handler. In our case, there are two possibilities: 0 (Kernel) or 3 (User), corresponding to the privileged and non-privileged levels respectively.

So, to handle all those exceptions, one call to `setInterruptHandler` for each exception has been added in the file `interrupt.c` (already implemented in function `set_handlers` in `libzeos.a`).

3.7 Programming interrupts

As expected the hardware interrupts work very similar to exceptions. Figure 13 shows the main steps for the clock interrupt. This interrupt may arrive at any time and deal with any point in the code. If the IDT has been correctly programmed, the `clock_handler` function programmed in the `entry.S` file will be executed. Inside this function we find a call to the `clock_routine` function that will make the final interrupt management. In the following sections we will show the required steps to program the keyboard interrupt and display the character corresponding to the pressed key.

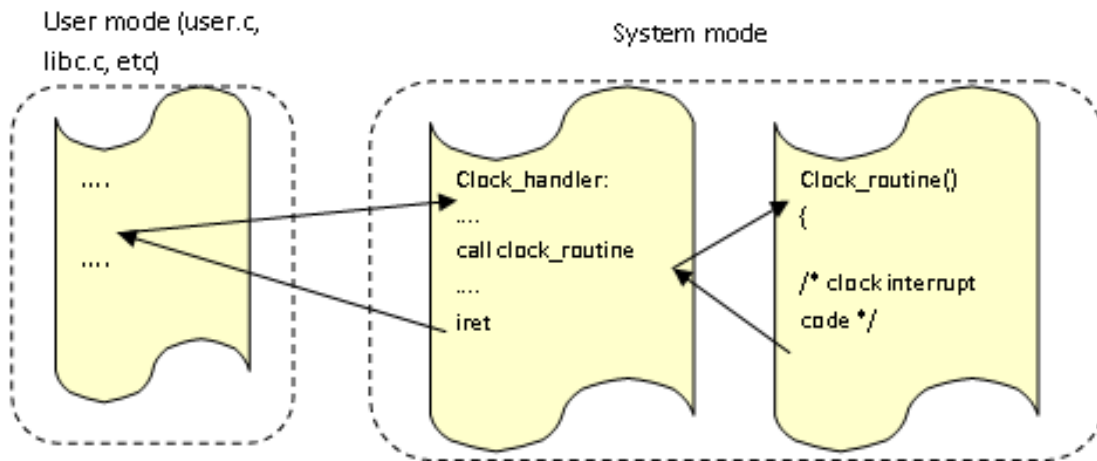


Fig. 13. Snapshot of the clock interrupt

3.7.1 The keyboard interrupt management

The keyboard interrupt will display the character that corresponds to the pressed key. Figure 14 shows the expected result with a C character displayed in a fixed place on the screen.

To program the keyboard interrupt management, you must:

- Write the service routine (`keyboard_routine`).
- Write the handler (`keyboard_handler`).
- Initialize the IDT as it is done for the exceptions. The keyboard interrupt is contained in entry 33.

```
setInterruptHandler(33, keyboard_handler, 0);
```

- Enable the interrupts. The key interrupt is a *masked* interrupt that is disabled in the provided code, which means that it is ignored by default. It is necessary to modify the mask that enables the interrupts, located in the `enable_int0` routine (file `hardware.S`).



Fig. 14. Display output after a keyboard interrupt, showing a capital C.

Notice that just after enabling the interrupts, any of them can be raised. When this occurs, you must ensure that the system is fully initialized because the interrupt service routines may access any system structure. For this reason, the best place to enable the interrupts is after all the OS services have been started and just before performing the return to user mode.

3.7.2 Writing the service routine

After pressing a key, the keyboard service routine will display on the screen the character that corresponds to the pressed key.

This service routine performs the following steps:

- 1) Read the port corresponding to the keyboard data register (0x60) with the routine of the *hardware.S* file:

```
unsigned char inb(int port)
```

- 2) Once the value is read from this port, it must distinguish between a make (key pressed) or a break (key released). The contents of this register is shown in Figure 15. Bit number 7 specifies whether it is a make or a break. Bits 0..6 contain the scan code to be translated into a character.
- 3) If it is a make, the translation table *char_map* at *interrupt.c* must be used to obtain the character that matches the scan code.
- 4) Print the character on the upper left of the screen. For this the *putc_xy* function in *io.c* is used.
- 5) If the pressed key does not match any ASCII character (Control, Enter, Backspace, ...) a capital C will be displayed.

3.7.3 Writing the handler

An interrupt handler is written more or less the same way as an exception handler. Follow the steps below:

- 1) Define an assembly header for the handler.
- 2) Save the context.

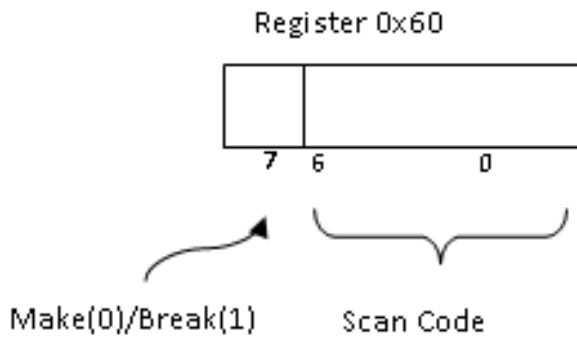


Fig. 15. Contents of the keyboard 0x60 register

- 3) Perform the **end of interruption (EOI)**. You must notify the system that you have received this interrupt and, therefore, you are ready to receive new interrupts. For this, a new macro, named EOI, has been created:

```
#define EOI \
movb $0x20, %al ; \
outb %al, $0x20 ;
```

- 4) Call the service routine.
- 5) Restore the context.
- 6) Return from the interrupt (be careful, since you are returning to user mode).

3.8 Programming system calls

In this section we will implement our first system call: the **write** system call. It enables the user application to print strings to the screen device. For more information, see Chapter 8 of *Understanding the Linux Kernel*.

As you have seen in other courses, system calls have a common entry point, in our case we will use the 0x80 interrupt. This single entry point (1 handler) will give access to multiple system calls. In this section we will present the common functions to all these system calls (and also the data structures) and the specific functions for the *write* call.

Figure 16 shows the steps followed by a system call, *write* in particular. The user code calls what it believes to be the system call (*write*), but it is actually just an adapter. The library code implements this adapter—which we will call a **wrapper**—that *wraps* 1) the **pass of parameters** between user and system modes, 2) the selection of the system service to execute, 3) the generation of the **trap** (int \$0x80) and 4) the processing of the result.

Interrupt 0x80 must be initialized in the IDT as usual. You should note that this is a software interrupt, meaning—in contrast to a hardware interrupt—that it is unmasked. Once the interrupt is generated, it is executed as any other interrupt (first the handler is executed and then the service routine). As explained above, all system calls have a common handler, which will be named **system_call**. In this handler, the system call service requested by the user is checked and the corresponding service routine is executed (**sys_write** in the example).

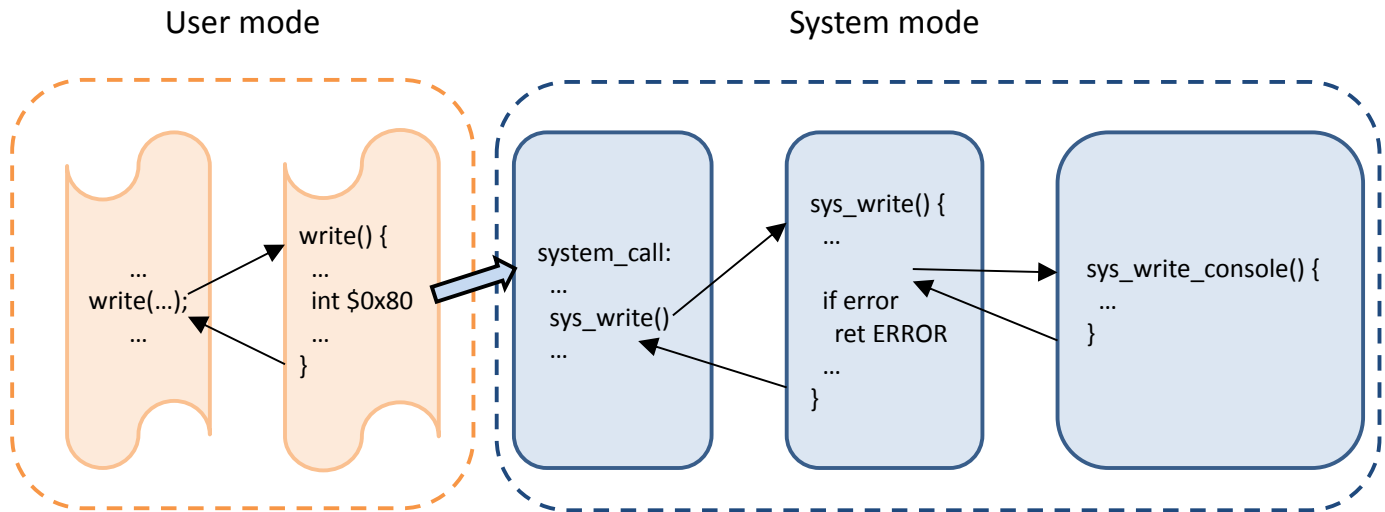


Fig. 16. Related functions and steps followed by the *write* system call.

3.8.1 Independence from devices

As explained in other courses, one of the basic principles of an OS design is independence between devices; this is, the interface for input and output system calls is the same regardless of the device that is requesting its service.

Keeping this in mind, you will design each call considering a part that is dependent and a part that is independent from the device. Thus, the system will be prepared to grow in complexity, or to easily add more devices to it. The design will be similar to a real OS.

For example, in this session the **sys_write** function (independent part) will call **sys_write_console** (dependent part), in order to print on the screen (but in the future it may call *sys_write_printer* or *sys_write_ext2* or ...).

3.8.2 Writing the write wrapper

The *write* wrapper header is:

```
int write (int fd, char * buffer, int size);
```

Wrappers must carry out the following steps:

- 1) Parameter passing (see Section 3.8.3).
- 2) Put the identifier of the system call in the EAX register (number 4 for write).
- 3) Generate the trap: *int \$0x80*.
- 4) Process the result (see section 3.8.4).
- 5) Return.

These wrappers **must** be coded in assembly.

3.8.3 Parameter passing

ZeOS implements the parameter passing from user mode to system mode through the CPU registers, as occurs in Linux. The parameters of the stack must be copied to the registers EDX,

ECX, EBX, ESI, EDI, EBP. The correct order is the first parameter (on the left) in EDX, the second parameter in ECX, etc. Note that this order is a hack to exploit the `SAVE_ALL` macro and to create the stack structure from section 3.5.

3.8.4 Returning results

The convention in Linux for returning the results of the system calls considers a positive or zero return when the execution of the call is correct and a negative value when an error is detected. In case of error, this negative value specifies the error.

Nevertheless, the returned value of the system call does not reach the user directly, but instead it is processed by the wrapper of the system call that is found in the library. Thus, if the returned value is positive or zero, it will be returned to the user as is. However, if the value is negative, the wrapper will save the absolute value of the return in an error variable defined in the library (called **errno**) and return -1 to the user to notify that the system call has an error. If the user requires further information about the error, he/she must consult the `errno` variable. If the system call does not return an error, the `errno` variable is not modified.

To use this convention in ZeOS, **all** system calls must return a negative number, that specifies the generated error in the event of a wrong execution. The constants used to identify these errors must be contained in an **errno.h** file (as in Linux). Moreover, your system call wrappers must process the negative values of the calls, update the `errno` variable with the absolute value of these errors and return -1 to the user.

A function should also be added for the users to obtain information about the error generated by the previous system call. This function have this interface:

```
void perror(void);
```

It will write an error message for the standard output according to the value of the `errno` variable.

Note: In the description of the system calls that appear in this document, the returned values are considered from the point of view of the user, which means that -1 will always be returned in the case of an error.

3.8.5 Service Routine to the write system call

In this section we will define the **sys_write** service routine, which checks that everything works correctly and shows the characters on the screen.

```
int sys_write(int fd, char * buffer, int size);
    fd: file descriptor. In this delivery it must always be 1.
    buffer: pointer to the bytes.
    size: number of bytes.
    return ' Negative number in case of error (specifying the kind of error) and
           the number of bytes written if OK.
```

System calls (in general) follow these steps:

- 1) **Check the user parameters:** fd, buffer and size. Bear in mind that the system has to be robust and assume that the parameters from the user space are unsafe by default (**lib.c** is user code).

- a) Check the fd, we will use a new **int check_fd (int fd, int operation)** function that checks whether the specified file descriptor and requested operation are correct. The operations can be **LECTURA** or **ESCRITURA**. If the operation indicated for this file descriptor is right, the function returns 0. Otherwise, it returns a negative identifier of the generated error.
 - b) Check buffer. In this case, it will be enough to verify that the pointer parameter is not **NULL**, but in the future you will need to check that it is a valid address.
 - c) Check size. Check positive values.
- 2) **Copy the data from/to the user address space if needed.** See the functions `copy_to_user` and `copy_from_user` (section 3.8.6).
 - 3) **Implement the requested service. For the I/O system calls, this requires calling the device dependent routine.** In this particular case, the device dependent routine is `sys_write_console`:

```
int sys_write_console (char *buffer, int size);
```

This function displays *size* bytes contained in the *buffer* and returns the number of bytes written.

- 4) **Return the result.**

3.8.6 Copying data from/to the user address space

Copying data from/to the user is a critical operation because it could be a cause of kernel vulnerability. During the project, even it is possible to access the different address spaces because they are disjoint, you are asked to use a couple of operations (`copy_from_user` and `copy_to_user`) to explicitly mark the data transfers between both memory address spaces.

The Linux Kernel Module Programming Guide argues the need for these functions as follows:

"The reason for `copy_from_user` or `get_user` is that Linux memory (on Intel architecture, it may be different under some other processors) is segmented. This means that a pointer, by itself, does not reference a unique location in memory, only a location in a memory segment, and you need to know which memory segment it is to be able to use it. There is one memory segment for the kernel, and one for each of the processes. The only memory segment accessible to a process is its own, so when writing regular programs to run as processes, there's no need to worry about segments. When you write a kernel module, normally you want to access the kernel memory segment, which is handled automatically by the system. However, when the content of a memory buffer needs to be passed between the currently running process and the kernel, the kernel function receives a pointer to the memory buffer which is in the process segment. The `copy_from_user` and `copy_to_user` functions allow you to access that memory".

This means that `copy_to_user` and `copy_from_user` encapsulate complexity due to processor architectural differences. In our scenario, these functions will only be useful for copying data between the user and the OS address space.

3.8.7 Writing the handler

The steps are as follows:

- Save the context.
- Check that it is a correct system call number (that the system call identifier belongs to a defined range of valid system calls). Otherwise, the corresponding error must be returned.
- Execute the corresponding system call.

- Update the context so that, once restored, the result of the system call can be found in the EAX register. Remember that C functions store their result in the EAX register²⁵.
- Restore the context.
- Return to the user mode.

A table is needed to relate the identifier of each call to its routine. The table will be filled in as new system calls are added. This call table, named **sys_call_table**, will be defined in assembly like:

```
ENTRY(sys_call_table)
.long sys_ni_syscall // sys_ni_syscall address (not implemented)
.long sys_functionname // sys_functionname address
```

Each entry contains the memory address of the function to be called. Bear in mind that non-implemented calls in a valid range must implement a call to **sys_ni_syscall**. This function will just return a negative identifier for the non-implemented call error. In this case the table defines a single system call with the identifier 1 that will call the *sys_functionname* (Note that identifier 0 will call the not-implemented function).

Thanks to Intel's memory addressing modes, a single instruction is needed to call a system function from this table indexed by a register (*EAX* in this case):

```
call *sys_call_table(, %EAX, 0x04);
```

So, the syscall table must be extended to enable the write system call. In particular, entries from 0 to 3 must be initialized to **sys_ni_syscall** to return the corresponding error (the syscall does not exist). The 5th entry (identifier 4) points to the service routine (that will be implemented in section 3.8.5):

```
ENTRY (sys_call_table)
    .long sys_ni_syscall    // 0
    .long sys_ni_syscall    // 1
    .long sys_ni_syscall    // 2
    .long sys_ni_syscall    // 3
    .long sys_write         // 4
.globl MAX_SYSCALL
MAX_SYSCALL = (. - sys_call_table)/4
```

- Visit https://elixir.bootlin.com/linux/v3.2.102/source/arch/x86/kernel/syscall_table_32.S to look at the Linux declaration of the **sys_call_table** table.²⁶

Finally, if the number of the syscall is outside the correct range (from 0 to 4 in this case), the **sys_ni_syscall** will be called. This function must always return the corresponding error:

```
int sys_ni_syscall()
{
    return -38; /*ENOSYS*/
}
```

25. If the context is not modified, the return value will be the system call identifier.

26. Since version 3.3 this table is automatically generated.

The code for the OS entry point (position 0x80 of the IDT) looks like:

```
ENTRY(system_call_handler)
    SAVE_ALL                // Save the current context
    cmpl $0, %EAX           // Is syscall number negative?
    jl err                 // If it is, jump to return an error
    cmpl $MAX_SYSCALL, %EAX // Is syscall greater than MAX_SYSCALL (4)?
    jg err                 // If it is, jump to return an error
    call *sys_call_table(, %EAX, 0x04) // Call the corresponding service routine
    jmp fin                // Finish
err:
    movl $-ENOSYS, %EAX     // Move to EAX the ENOSYS error
fin:
    movl %EAX, 0x18(%esp)   // Change the EAX value in the stack
    RESTORE_ALL            // Restore the context
    iret
```

3.8.8 IDT initialization

Finally, initialize the entry point of a system call using the call:

```
void setTrapHandler(int posicio, void (*handler)(), int nivellPriv)
```

This function is similar to `setInterruptHandler`. In this case, since the system calls are invoked from the user privilege level, the value for the third parameter must be 3:

```
setTrapHandler(0x80, system_call_handler, 3);
```

3.9 Programming fast system calls

Modern processors extend the way of invoking the operating system code with an additional mechanism to enter the system: *fast system calls*. The main difference of those calls compared to software interrupts are:

- 1) The IDT is not accessed. Software interrupts access the IDT to find the operating system entry point address. Fast calls do not use the IDT because the entry point address is obtained from a specific register of the processor, called Model Specific Register(MSR)²⁷
- 2) No check of the privilege level is performed. Fast calls can only be executed from user privilege level. The operating system code does not use them. So, there is no need to check the privilege level since it will always be executed from user mode.
- 3) The TSS is not accessed to find the address of the system stack. Instead, this value is read from another MSR.
- 4) Fast calls do not store the processor's state nor the minimal information to return to user mode in the stack.

The fast syscall mechanism is based on the pair of assembly instructions *sysenter* and *sysexit*.

27. The Model Specific Registers are additional registers of the processor. They can not be used as general purpose registers because they have a specific meaning related to the operating system and they enable different features of the processor.

sysenter is used to enter the system. It uses specific Model Specific Registers (MSRs) to find the entry address of the operating system and the system stack to use. In particular it needs:

MSR		Use
SYSENTER_CS_MSR	(0x174)	Operating system code segment (CS)
SYSENTER_ESP_MSR	(0x175)	Operating system stack (ESP)
SYSENTER_EIP_MSR	(0x176)	Operating system entry point, the handler (EIP)

These registers can be accessed and modified using the assembly instructions *rdmsr* and *wrmsr*. They must be set up during the initialization of the system.

sysexit is used to switch back to user mode. *sysexit* relies on the fact that register EDX contains the address of user code to return back and ECX points to the top of the user stack. So, *sysexit* **does not access at all** the system stack to find those addresses as happens with the software interrupts.

3.9.1 Writing the wrapper

A wrapper for performing a fast system call is very similar to that of Section 3.8 with a couple of differences:

- 1) Instead of using the assembly instruction *int*, it must use *sysenter* to invoke the operating system code.
- 2) Since *sysenter* does NOT store anything in the system stack and *sysexit* requires to know where to return back (EIP and ESP), we need to store this information somewhere else. In particular we need to know the position in the wrapper code (EIP) and the position in the stack (ESP) used in user mode before entering into system mode. Therefore, in the wrapper we will push these values on the top of the user stack, and use register EBP to easily find them from system mode—simulating a false function call and the initial *dynamic link*.

So, the steps of a wrapper using *sysenter* are:

- 1) Parameter passing. Similar to that in Section 3.8.
- 2) Put the identifier of the system call in the EAX register.
- 3) Store the return address to user mode in the user stack. This address corresponds to the assembly instruction following the *sysenter* instruction. You can use a label to mark the next instruction, and store this address in the stack.
- 4) Create a fake dynamic link (pushing EBP and storing current value of ESP into EBP). This makes the EBP register point to the top of the user stack and it can be used from the handler in system mode to access any user value easily.
- 5) Enter the system: *sysenter*.
- 6) Process the result.
- 7) Return.

3.9.2 Writing the handler

sysenter does not store the user mode information in the system stack. This makes that a different handler for *sysenter* is needed in the operating system since — as explained in section 3.5 — it must construct the stack in a specific way to ensure that this stack will be fully compatible with the service routines. In addition, to switch back to user mode, the handler executes *sysexit*, setting EDX and ECX to the correct values.

This new handler has the following steps:

- 1) Save in the stack the same user mode information than the *int* instruction stores when switching to system mode (as explained in section 3.5). In particular, the user return address and the position from the user mode stack, both accesible through the register EBP.
- 2) Save the context.
- 3) Check if it is a correct system call number (EAX).
- 4) Execute the corresponding system call (service routine).
- 5) Update the context to store the returned value.
- 6) Restore the context.
- 7) Return to user mode using *sysexit*:
 - Set EDX with the user return address from the stack.
 - Set ECX with the value of the user EBP from the stack.
 - Invoke *sysexit*.

With this procedure, all structures (the syscall table) and code (service routines) will be fully compatible in both *int* and *sysenter* mechanisms.

The handler for *sysenter* looks like:

```
ENTRY(syscall_handler_sysenter)
    push $__USER_DS
    push %EBP          // User stack address
    pushfl
    push $__USER_CS
    push 4(%EBP)        // User return address
    SAVE_ALL
    cmpl $0, %EAX
    jl sysenter_err
    cmpl $MAX_SYSCALL, %EAX
    jg sysenter_err
    call *sys_call_table(, %EAX, 0x04)
    jmp sysenter_fin
sysenter_err:
    movl $-ENOSYS, %EAX
sysenter_fin:
    movl %EAX, 0x18(%ESP)
    RESTORE_ALL
    movl (%ESP), %EDX    // Return address
    movl 12(%ESP), %ECX  // User stack address
    sti                 // Enable interrupts again
    sysexit
```

3.9.3 Initializing fast system calls

Since *sysenter* does not use the IDT to find the address of the system call handler but uses the MSRs of the processor, the first thing to do is to enable the *sysenter* entry point. For this:

- 1) Write an **assembly** function called *writeMSR* that accepts two parameters: the number of the MSR and the value to store in that MSR. Note: check the syntax of *wrmsr* in the Intel manuals to know the type and the number of parameters of this assembly instruction.

- 2) Use the previous function, after the IDT initialization, to program the MSR. In particular, MSR 0x174 must be set to `KERNEL_CS`, 0x175 to `INITIAL_ESP` and 0x176 to the address of the operating system *sysenter* handler.

3.10 Work to do

In this session you need to:

- 1) Complete Zeos code.
- 2) Implement the keyboard management.
- 3) Implement the *write* system call.
- 4) Implement the clock management.
- 5) Implement the *gettime* system call.
- 6) Implement the page fault exception management.

3.10.1 Complete Zeos Code

The released code lacks some features explained in the ZEOS document. You have to **complete** them:

- Implement the macro **RESTORE_ALL**.
- Implement the macro **EOI**.

3.10.2 Implement the keyboard management

We want to show the key pressed by the keyboard, therefore we need to manage keyboard interrupts:

- Implement the keyboard service routine.
- Implement the keyboard handler.
- Initialize the IDT with the keyboard handler
- Enable the interrupt.

3.10.3 Implement the write system call.

We want to allow user code to show messages to the screen, therefore implement a new system call *write*:

- Implement the *sys_write* routine.
- Modify the *sys_call_table* with the new routine.
- Create a wrapper for the system call.
- Implement the *system_call_handler* routine.
- Initialize either the IDT or the MSRs (or both) with the handler, depending on the mechanism to invoke the operating system code used (*int*, *sysenter* or both).
- Implement the *errno* and *perror* function.

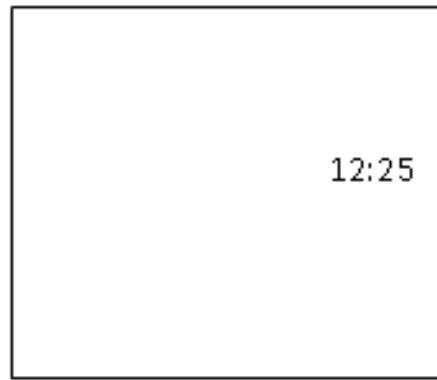


Fig. 17. Clock interrupt prints a clock with the minutes and seconds elapsed.

3.10.4 Clock management

The clock management will display the seconds that have elapsed since the boot process. Figure 17 shows the expected result with the time displayed in a fixed place on the screen.

To write the management of an interrupt such as the clock interrupt (that is, a masked type), you must:

- Write the service routine.
- Write the handler.
- Initialize the entry of the clock interrupt in the IDT. The clock interrupt is contained in entry 32.
- Enable the interrupt. The clock interrupt is a masked interrupt that is disabled in the code provided, which means that it is not dealt with. Modify the mask that enables the interrupts, located in the **enable_int()** routine (file **hardware.S**).

Notice that just after enabling the interrupts, one of them can be raised at any time. When this occurs, you must ensure that the system is fully initialized because the interrupt service routines may access any system structure.

Write the clock service routine

Write the corresponding service routine of the clock interrupt. Inside this routine, you must call: **zeos_show_clock** which will display at the top-right part of the screen a clock with the elapsed time since the OS booted. Notice that in bochs the clock interrupt is also emulated. So, this showed time goes faster than the real one.

The header of this function is located at **zeos_interrupt.h**:

```
void zeos_show_clock();
```

Write the clock handler

An interrupt handler is written more or less the same way as an exception handler. Follow the steps below:

- 1) Define an assembly header for the handler.
- 2) Save the context.
- 3) Perform the EOI (remember that there is a macro for this available). You must notify the system that you are treating the interrupt.
- 4) Call the service routine.
- 5) Restore the context.
- 6) Return from the interrupt to user mode.

Test the clock

The seconds that have elapsed since the OS boot process must appear on the screen. If it does not work, determine the problem with the debugger. Check whether the interrupt is executed and the handler is executed, the interrupt routine is executed, whether you return to the user mode, etc.

3.10.5 Gettime system call

Extend ZeOS to incorporate this new system call. This syscall returns the number of clock ticks elapsed since the OS has booted. Its header is:

```
int gettimeofday();
```

To implement this system call you will:

- 1) Create a global variable called `zeos_ticks`.
- 2) Initialize `zeos_ticks` to 0 at the beginning of the operating system (main).
- 3) Modify the clock interrupt service routine to increment the `zeos_ticks` variable.
- 4) Write the wrapper function. The identifier for this system call will be 10.
- 5) Update the system calls table
- 6) Write the service routine.
- 7) Return the result.

3.10.6 Manage Page Fault exceptions

When we program, a usual problem is that we access an invalid memory address. This access generates a “Page Fault” exception depending on the type of access. You may check it right now adding the following lines to the user code:

```
char* p = 0;
*p = 'x';
```

Which tries to write character ‘x’ to the memory address ‘0’ which is not present and, therefore, it raises a Page fault exception.

It is useful to know which code generates the exception (the specific memory address), and therefore we want you to reprogram the page fault exception to show a message in screen like:

```
Process generates a PAGE FAULT exception at EIP: 0xXXXXXX
```

Where '0XXXXXX' is the memory address corresponding to the EIP that generated the exception. This address must be extracted from the saved hardware context of the exception. After printing the message, you must enter an infinite loop.

4 ACKNOWLEDGEMENTS

This document was drawn up with the support of professors on previous courses: Julita Corbalán, Marisa Gil, Jordi Guitart, Gemma Reig, Amador Millán, Jordi García, Silvia LLorente, Pablo Chacín and Rubén González. The authors wish to thank A. Bartra, M. Muntanyá and O. Nieto for their contributions.

This document has been improved by the following people:

- Albert Batalle Garcia (2011)

APPENDIX

Instructions to prepare the development environment:

- 1) Install basic development packages (*gcc*, *ld*, ...)

```
sudo apt-get install build-essential
```

- 2) Install package for x86 tools (*as86* and *ld86*)

```
sudo apt-get install bin86
```

- 3) Install needed libraries

```
apt-get -y install libx11-dev
apt-get -y install libgtk2.0-dev
apt-get -y install libreadline-dev
```

- 4) Download and install Bochs 2.6.7. You need to compile the Bochs source code in order to use the Bochs debugging facilities. It is necessary to activate some debugger options during the compilation because the option to use the debugger is disabled by default in the Bochs Binary Standard Package.

To enable the external GDB debugger you need to execute *configure* with parameter *-enable-gdb-stub*, and recompile Bochs:

```
$ ./configure --enable-gdb-stub --with-x --prefix=/opt/bochs_gdb
$ make all install
```

If you want to activate the internal debugger you must recompile Bochs using:

```
$ ./configure --enable-debug --enable-disasm --enable-x86-debugger \
--enable-readline --with-x --prefix=/opt/bochs
$ make all install
```