

Zeos

Yolanda Becerra, Juan José Costa and Alex Pajuelo

Course 2019–2020



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 (assembler) 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:

- Input/output mechanisms (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	6
2	INTRODUCTORY SESSION	6
2.1	Working environment	6
2.2	Getting started	7
2.3	Introduction to ZeOS	7
2.3.1	ZeOS source code files	8
2.3.2	ZeOS binary image	9
2.4	Boot process	10
2.5	Image construction	11
2.5.1	Bochs	11
2.5.2	Debugging using Bochs	12
2.5.3	Frequently used debugging commands	15
2.6	Work to do	15
2.6.1	Initial steps: Understanding what is being executed	16
2.6.2	User code modification	16
2.6.3	Use of assembler	18
2.6.4	System code modification	19
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	25
3.6.1	Writing the service routines	25
3.6.2	Exception parameters	26
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	29
3.8	Programming system calls	29
3.8.1	Independence from devices	30
3.8.2	Returning results	30
3.8.3	Writing the write wrapper	31
3.8.4	Service Routine to the write system call	31
3.8.5	Copying data from/to the user address space	32
3.8.6	Writing the handler	32
3.8.7	IDT initialization	34
3.9	Programming fast system calls	34
3.9.1	Writing the wrapper	35
3.9.2	Writing the handler	36
3.9.3	Initializing fast system calls	36
3.10	Work to do	37
3.10.1	Complete Zeos Code	37
3.10.2	Clock management	37
3.10.3	Gettime system call	39
4	BASIC PROCESS MANAGEMENT	40
4.1	Prior concepts	40
4.2	Definition of data structures	40
4.2.1	Process data structures	40
4.2.2	Process identification	43
4.3	Memory management	43
4.3.1	MMU: Paging mechanism	44
4.3.2	The directory and page table	44
4.3.3	Translation Lookaside Buffer (TLB)	46
4.3.4	Specific organization for ZeOS	47
4.4	Initial processes initialization	50
4.4.1	Idle process	50
4.4.2	Init process	51
4.4.3	Zeos implementation details	52
4.5	Process switch	53
4.5.1	Initial testing	54
4.6	Process creation and identification	54

4.6.1	<i>getpid</i> system call implementation	54
4.6.2	<i>fork</i> system call implementation	54
4.7	Process scheduling	56
4.7.1	Main scheduling functions	56
4.7.2	Round robin policy implementation	57
4.8	Process destruction	57
4.9	Statistical information of processes	58
4.10	Work to do	59
5	BLOCKING OPERATIONS AND PERFORMANCE EVALUATION	60
5.1	Support to different scheduling policies	60
5.1.1	What is already done	60
5.1.2	What do you have to do	61
5.2	Support to consider blocked processes	61
5.2.1	What is already done	62
5.2.2	What do you have to do	62
5.3	Experiments	63
5.3.1	What do you have to do	63
5.3.2	Some issues to consider when defining the workloads	64
6	PROCESS COMMUNICATION	64
6.1	Performance evaluation	65
6.2	Server schemes to implement and to evaluate	65
6.3	Submission	66
7	ACKNOWLEDGEMENTS	66
	Appendix	66

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 assembler language.
- Add assembler 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, you need a working environment usually offered by another operating system. In this case, to develop ZeOS we will use an Ubuntu system with the following configuration:

- OS. Ubuntu 16.04¹
- Compiler. GCC 4.x
- Emulator. Bochs version 2.6.7 (bochs.sourceforge.net)

This environment will enable you to generate the ZeOS operating system. 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 system will be recompiled and loaded many times, so it is advisable to run your system on an architecture emulator (like *Bochs*) in order to save time. This way, when modifications are necessary, only the emulator has to be rebooted rather than the 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:

```
$ 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.).

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

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 16.04:

- 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 xzf 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:

```
$ 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. <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).

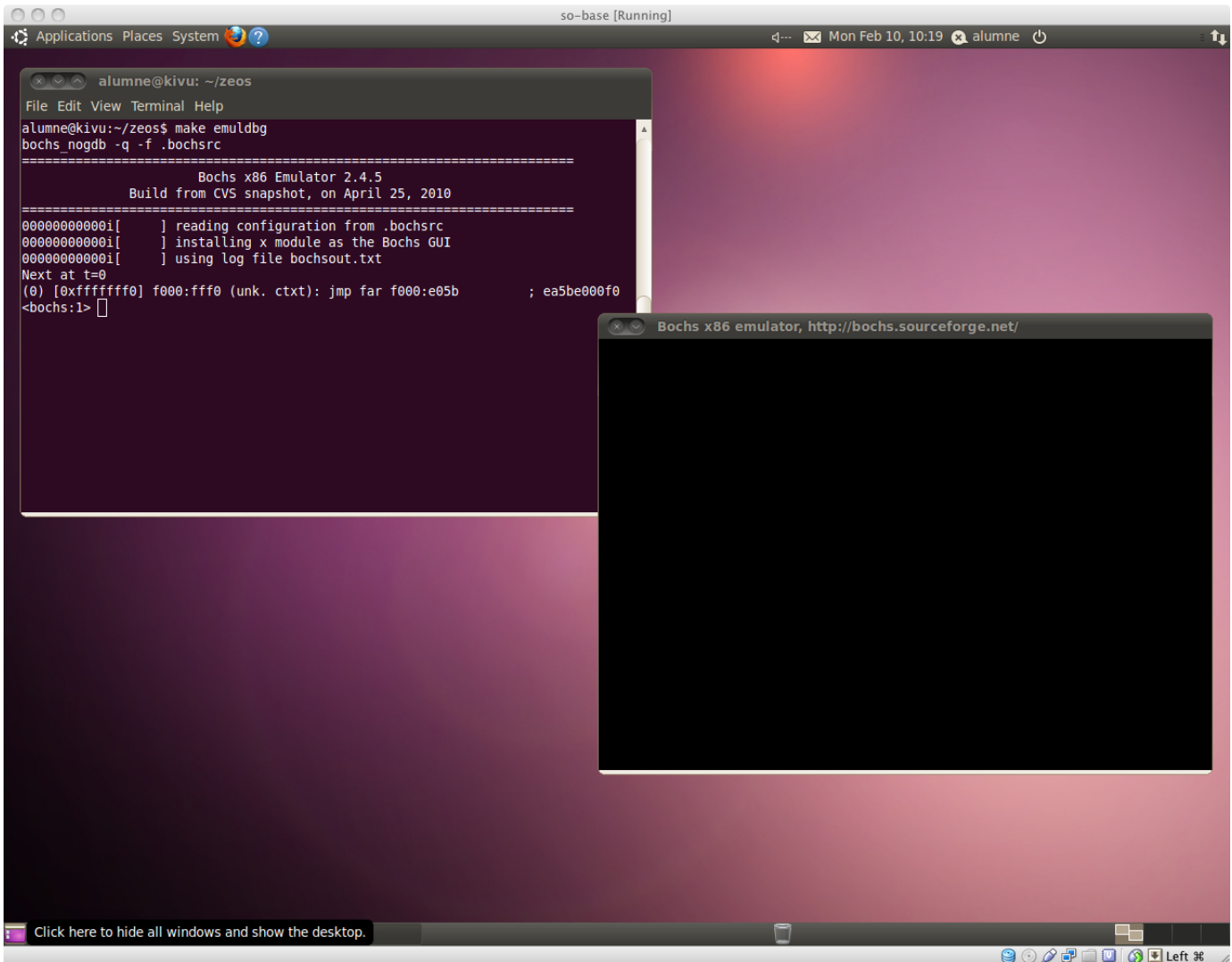


Fig. 1. Working environment with the Bochs commands window (left) and Bochs emulation window (right)

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 assembler 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.
- `.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.

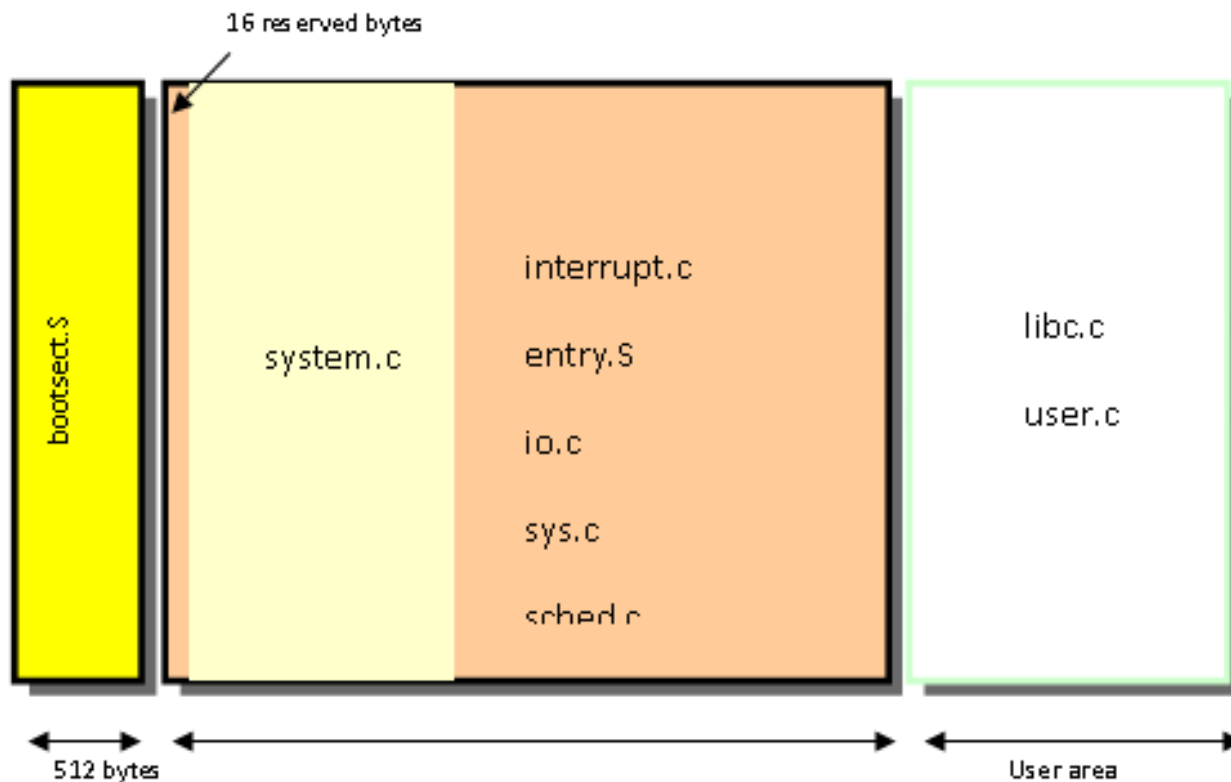


Fig. 2. The three main blocks (bootloader, system and user) forming the ZeOS binary image.⁴

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 divided in three main blocks (as shown in Figure 2):

- 1) The **boot sector block**. This block only comprises one file (*bootsect.S*) corresponding to the bootloader. It must weigh exactly 512 bytes, since it has to fit in a single sector.
- 2) The **system area block**. This block contains the main files of the 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 is 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.

One may ask why the user program is attached to the OS when this actually never happens. User programs are usually found on a hard disk inside a file system in a directory of this file system. The simple answer is that currently there is not either file system or an executable loader. Therefore, the user program is attached to the OS (in a position that is easy to calculate), and it is copied from its location in the image to a memory area with user privileges to which execution is transferred.

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

```

ENTRY(main)
SECTIONS
{
    . = 0x10000; /* system code begins here */
    .text.main :
    {
        BYTE(24); /* reserved to store user code management data */
        *(.text.main) }
    .text : { *(.text) }
    .rodata : { *(.rodata) }
    .data : { *(.data) } /* Initialized data */
    .bss : { *(.bss) } /* Not initialized data */
    . = ALIGN(4096); /* task_structs array aligned to page */
    .data.task : { *(.data.task) }
}

```

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

```

union task_union task[NR_TASKS] __attribute__((__section__(".data.task")));

```

Fig. 4. Task_struct array annotated with the section .data.task (sched.c)

Figure 3 shows the contents of the system.lds file that enables the linker to locate on the memory the various sections of an executable file. In this case, it sets the starting address (0x10000) of the system code; reserves 24 bytes, using the BYTE directive, corresponding to 3 integers to store the system image size, the user image size and some extra bytes reserved for future use; 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.task) which will contain the task_struct array.

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).

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.

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 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 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. First it moves the user area to the unprivileged memory area (starting at 0x100000), and finally it transfers the execution to this area, starting the execution of the unprivileged code.

2.5 Image construction

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

The program that attaches the files is called *build* (*build.c*). This program is generated (by the *make* command) 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 (make does this for you):

```
$/build bootsect system.out user.out > zeos.bin
Boot sector 512 bytes.
System is 24 kB
User is 1 kB
Image is 25 kB
```

Where *bootsect* is the binary content of the boot sector; *system.out* is the binary content of the system area; *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 finish the operating system load in memory. Once in memory, the OS code must move the user code to its final position, the user code start, 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.

2.5.1 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**.

Execution

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 a 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
```

2.5.2 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 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`) and use the Bochs internal debugger (`bochs_nogdb`) only when everything else fails.

Controlling execution through internal Bochs debugger

To start the Bochs internal debugger you can use:

```
$ make emuldbg
```

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

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

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 "-g" flag in CC (it 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 assembler instruction, *'jmp far f000:e05b'* in this case). A summary of the commands that can be executed may be found at <http://bochs.sourceforge.net/doc/docbook/user/internal-debugger.html>.

Controlling execution through an external GDB debugger

In order to start the GDB debugger you can use:

```
$ make gdb
```

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 2.5.3 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 "-g" flag in CC (it 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

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>

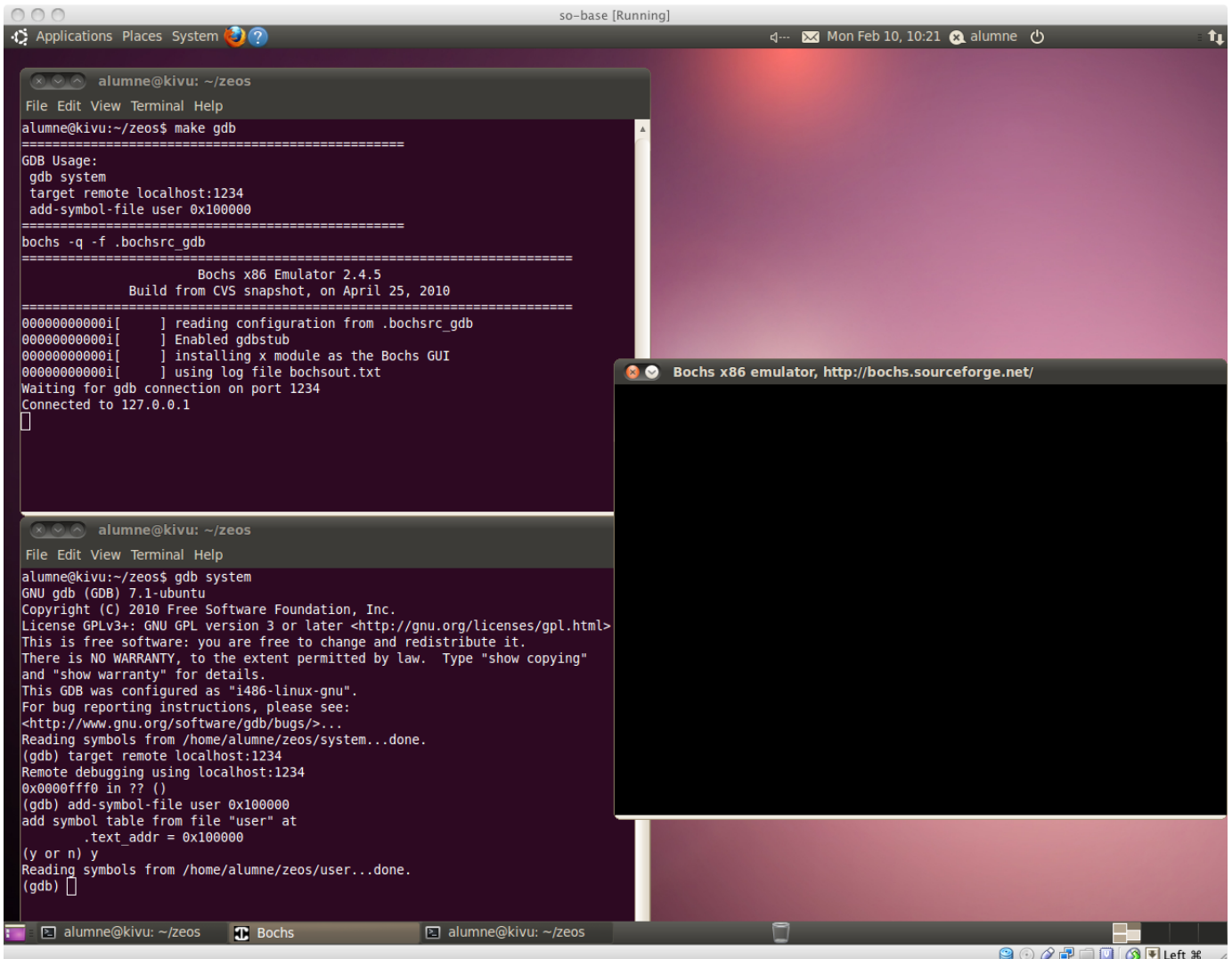


Fig. 5. Bochs commands windows (up), Bochs emulation window (right) and GDB window (down) after establishing a connection between GDB and Bochs.

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
```

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

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

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

2.5.3 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

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 assembler 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 assembler 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.
break 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>).
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)	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 paging address translation.
quit	quit	Exit debugger.

TABLE 1
Frequently used commands for Bochs and GDB.

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:

11. Like Beej's Quick Guide to GDB at <http://beej.us/guide/bggdb/>.

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

- 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.
- Mix assembler and C routines in your 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.
- 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 them) 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 assembler 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 its location in the code section of the generated image. For that, the commands *objdump* and *nm* can be used¹⁴.
 - The *objdump* command shows the compiler-generated assembler code and their corresponding memory addresses, which can be useful to insert breakpoints (Figure 6 shows the result of using *objdump* with the *system* file, where the C function *main* and its content are shown).
 - The *nm* command shows the variables and functions present in an object file and their memory locations. For instance, Figure 7 shows that, for example, there is an initialized variable (D) *task* (the task struct array) 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:

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*.

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


```

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    $0xfffffffff0, %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

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

- Modify the *main()* of the *user.c* file to add a call to the previous function with some dummy values (like 0x42 and 0x666) 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.¹⁶
- Execute the command "*objdump -d user.o*" to see the generated assembler code of the file *user.c*. Note that the addresses begin at zero.
 - Can you see the generated code of the function *add*?
- Now execute "*objdump -d user*". Do the addresses match? 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?
- Re-execute the program with breakpoints. To do this, at the debugger prompt set a BREAK-POINT at the first instruction of the *main* routine and another one at the *add* routine. Now

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, so after this session try to remove them all.

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.

- Take a look at the disassemble code from *user* file again. Can you see the invoking call to the function *add*?. Take a look at the flags used to generate this file and try to explain this behavior. Can you solve the problem?
- After solving the previous issue. Re-execute a part of the program step by step. To do this, set the breakpoint at the *add* routine and when arrived, use the commands *step* and/or *next* to move the execution forward¹⁷ and examine the memory content, the stack or the registers.

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 assembler

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

- Examine the generated assembler code for the *add* function and its invocation.
 - How is the dynamic link generated?
 - How are the parameters accessed?
 - How is the result returned?
- Create a new function *addAsm* in assembler 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 assembler code: *suma.S*.
 - Define the function *addAsm* using the *ENTRY* macro¹⁸ (defined in *include/asm.h*). This macro helps to define a function header so in this case:

```
#include <asm.h>

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

- 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 *entry.o* target).
- Add a call to the new assembler function *addAsm* in the *main* function of the *user.c* file by putting the result of the addition in a local variable. As the function is in a different file, we must instruct the C compiler to recognize the assembler function by adding a header for it. This header gives the compiler all the information it needs (number and type of the

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*.

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

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 assembler version works correctly.
- 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 (like your group identifier) to be printed at the beginning of your OS.
- Find the *printk* code and try to understand what is it doing.¹⁹ The screen device is a memory mapped device. In particular, the text mode offered by this device consists on a matrix 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.
- 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.

19. Some information about the screen device may be found online, for example at Chapter 23 of *The Art of Assembly* from Randall Hyde (<http://www.plantation-productions.com/Webster/www.artofasm.com/DOS/ch23/CH23-1.html#HEADING1-0>).

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 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 assembler 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 system stack from the Task State Segment (TSS) through the SS and the SP fields. Then the hardware saves in this new stack: 1) the current content of the SS and ESP registers (they point to the current user stack in this case), 2) the value of the EFLAGS (the cpu state word), and 3) the contents of CS and EIP registers (they contain the address that generated the interrupt). The resulting stack is shown in Figure 8.
- Execute the code at the address saved in the i^{th} entry of the IDT.

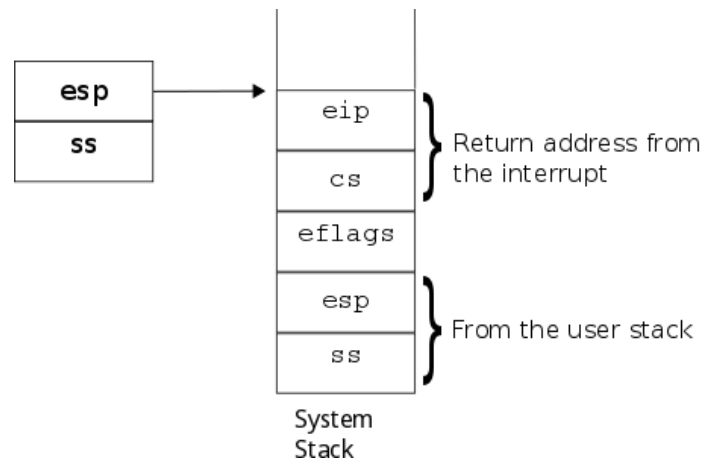


Fig. 8. State of the system stack when the execution of the interrupt handler begins

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 9 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.

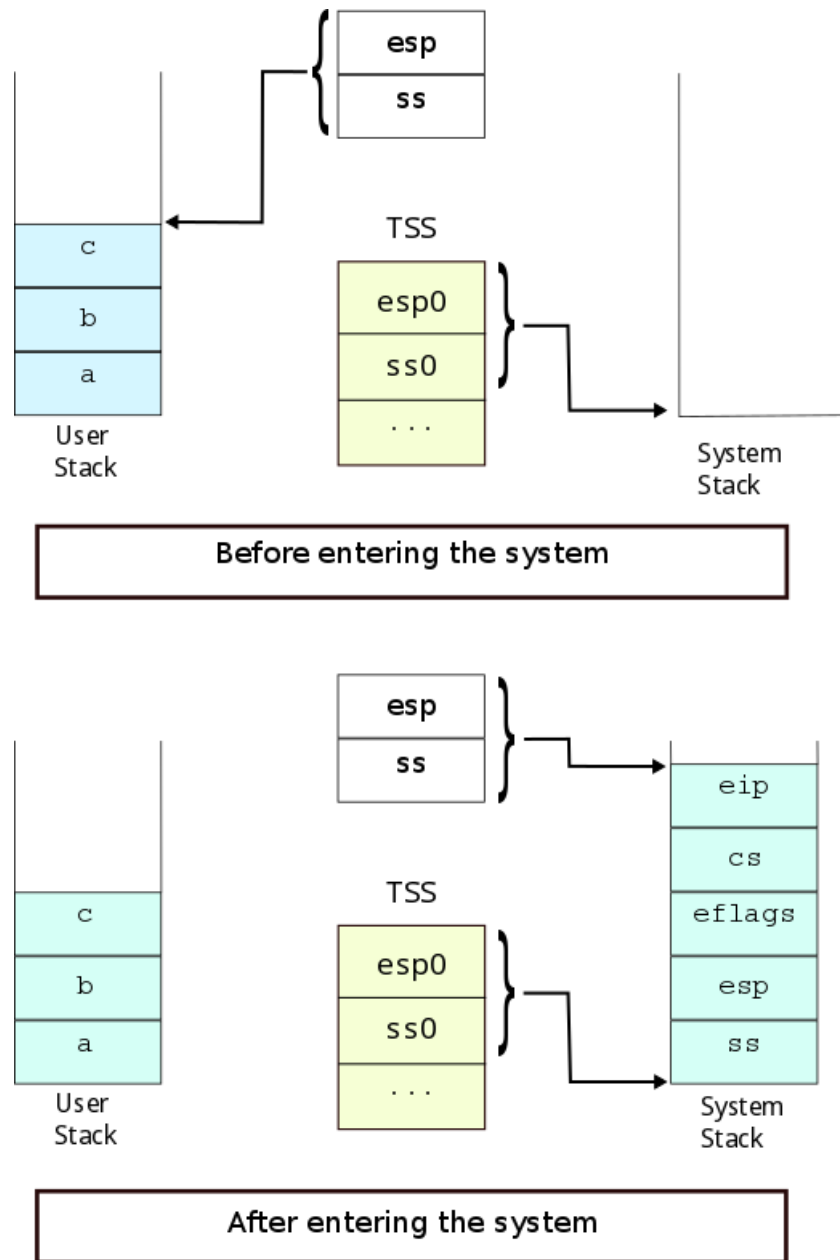


Fig. 9. The esp0 and ss0 fields of the TSS show the location of the system stack used in switch from user mode to system mode

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 10.

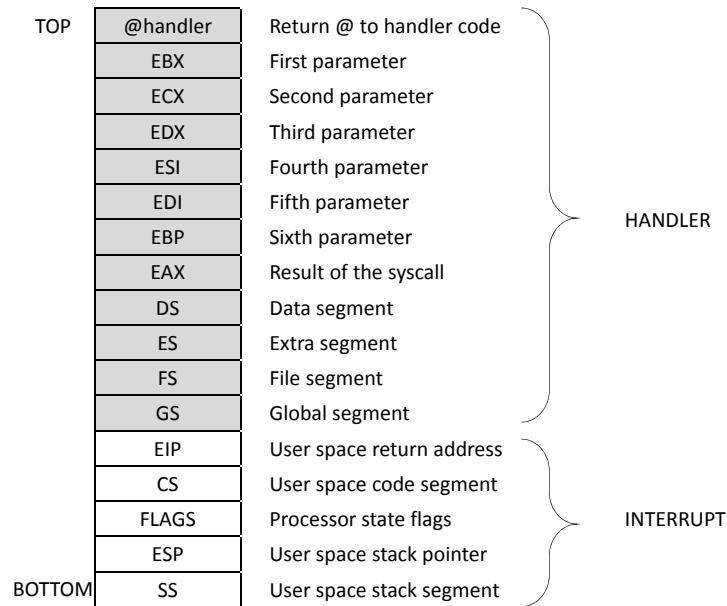


Fig. 10. Zeos system stack content at any system service routine after being called by the handler

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 10). 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 10). 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 code²⁰) 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.

20. 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.

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 11 shows the system stack contents and the location pointed by the *esp* and *ss* registers when an exception with an error parameter is raised.

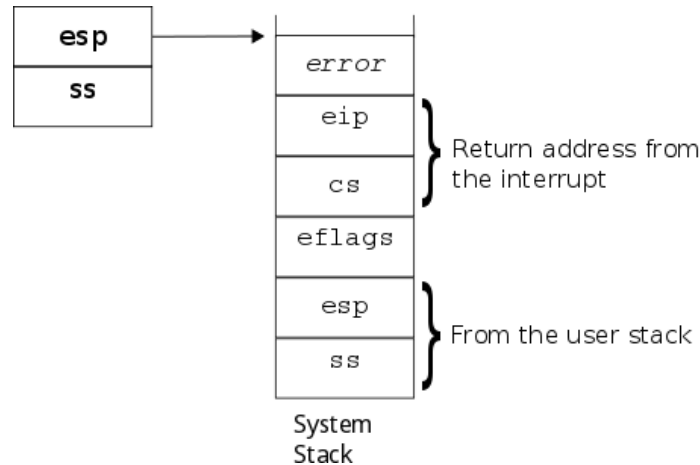


Fig. 11. System stack contents when an exception with an error parameter is raised

3.6.3 Writing the handler

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

- 1) Define the function header in assembler. The **ENTRY** macro²¹ can be used passing the name of the exception handler as a parameter (**ENTRY(handler_name)**).
 - From this point on, the **ENTRY** macro will always be used to define a function header in the assembler.
- 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. Since it is not a "normal" return because it has to change mode, an **iret** rather than a **ret**²² will be used for the return.

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:

21. Check the preprocessor document from the references to know how the macro mechanism of the assembler works.

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

```
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 exception²³.
- `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 12 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.

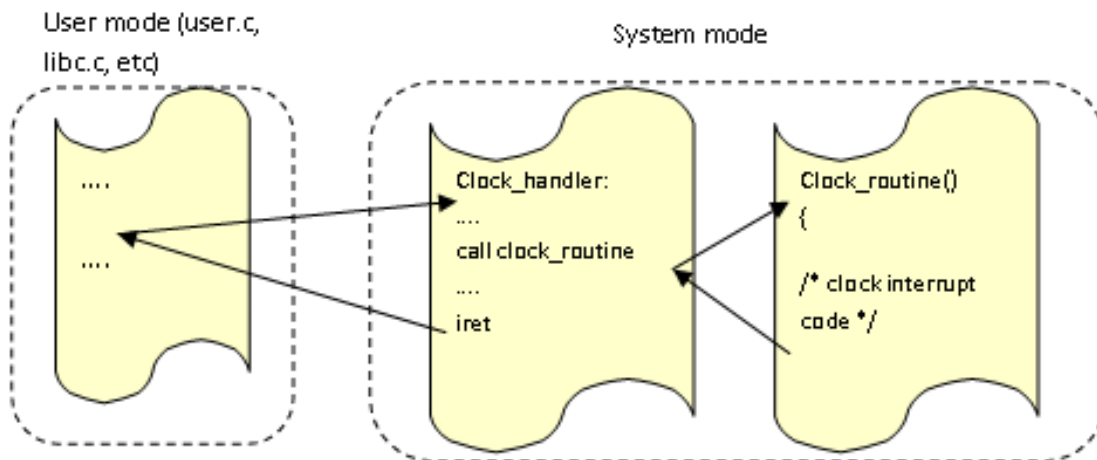


Fig. 12. Snapshot of the clock interrupt

3.7.1 The keyboard interrupt management

The keyboard interrupt will display the character that corresponds to the pressed key. Figure 13 shows the expected result with a C character displayed in a fixed place on the screen.

To program the keyboard interrupt management, you must:

²³. This is a *function pointer*, you may consult the K&R book "The C Programming language" to refresh your knowledge about them.



Fig. 13. Display output after a keyboard interrupt, showing a capital C.

- 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.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 *io.c* 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 14. 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.

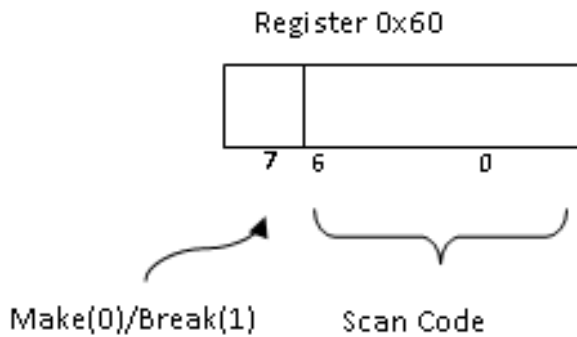


Fig. 14. Contents of the keyboard 0x60 register

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 assembler header for the handler.
- 2) Save the context.
- 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 15 shows the steps followed by a system call. The user code calls what it believes to be the system call, but it is actually only an adapter. The library code implements the adapter,

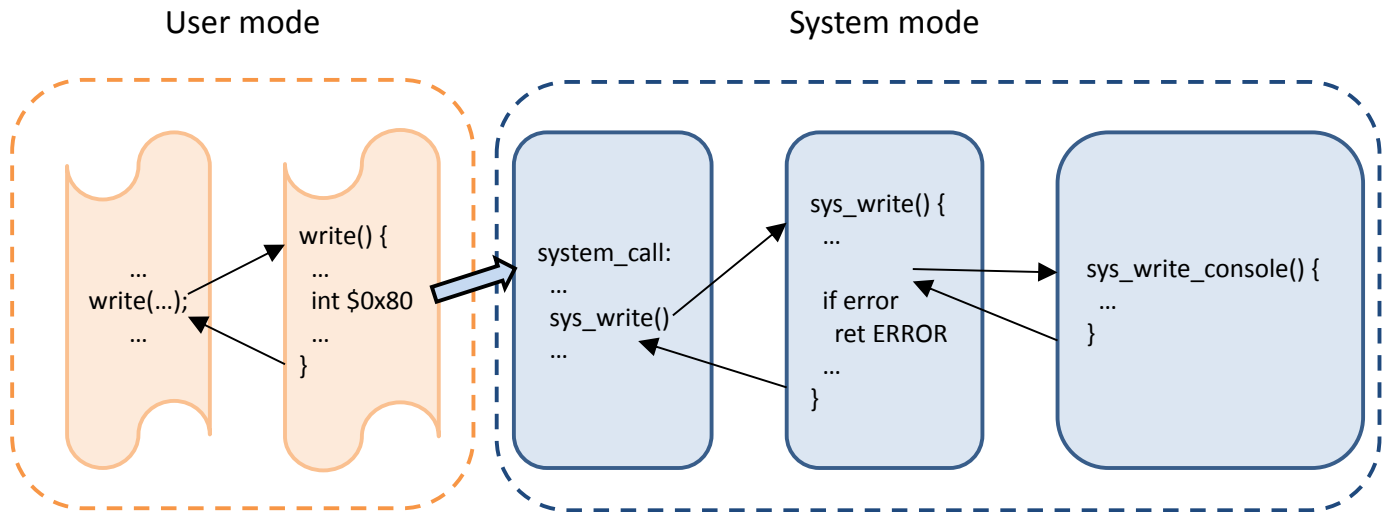


Fig. 15. Related functions and steps followed by the `write` system call.

which we will call a **wrapper**, because it *wraps* the **pass of parameters** between user and system modes, the generation of the **trap** (int `$0x80`) and the processing of the result.

In this case, interrupt `0x80` does not need to be enabled since it is unmasked. However, the IDT must be initialized. Once the interrupt is generated, it is executed as any other interrupt (the execution of the handler and the execution of 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).

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** call (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 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 and specify the generated error in the event of a wrong execution. The constants used to identify these errors are contained in the Linux **errno.h** file. 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.

3.8.3 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: 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 EBX, ECX, EDX, ESI, EDI. The correct order is the first parameter (on the left) in EBX, 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.
- 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 3.8.2).
- 5) Return.

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

3.8.4 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 `ESCRIPTURA`. 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`.
 - 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 3.8.5).
- 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.**

3.8.5 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.

3.8.6 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 register²⁴.
- 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 assembler 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*.

The instruction to execute the system call from this table indexed by the register *EAX* is:

```
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 3.8.4):

```
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.²⁵

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()
{
```

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

25. Since version 3.3 this table is automatically generated.

```

    return -38; /*ENOSYS*/
}

```

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?
    jnl 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

```

3.8.7 IDT initialization

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

```
void setTrapHandler(int posició, 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 will 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 stored in one of the specific registers of the processor, called Model Specific Registers(MSRs)²⁶
- 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 what is the privilege level from which the fast call is performed since it will be always 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.

26. 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 enable different features of the processor.

- 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 assembler instructions *sysenter* and *sysexit*.

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

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 assembler instructions *rdmsr* and *wrmsr*.

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 assembler instruction *int*, it must use *sysenter* to invoke the operating system code.
- 2) Since EDX and ECX registers will be modified by *sysexit* when returning back to the wrapper from the handler, the wrapper must ensure that the initial values of those registers are restored before leaving the wrapper.
- 3) The *sysexit* in the handler must be able to find the position in the wrapper (EIP) and the position in the stack (ESP) used in user mode to return back there. Therefore, these values will be pushed on the top of the user stack, keeping a pointer to it (EBP) for easily finding them in system mode.

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 values of ECX and EDX in the user stack.
- 4) Store the return address to user mode in the user stack. This address corresponds to the assembler instruction following the *sysenter* instruction. You can use a label to mark the next instruction, and store this address in the stack.
- 5) Create a fake dynamic link (pushing EBP and storing current value of ESP into EBP). This allows to use the EBP register from the handler in system mode to access the top of the user stack and from there access any user value easily.
- 6) Enter the system: *sysenter*.
- 7) Remove the temporal data: Pop EBP, the return address, EDX and ECX from the stack.
- 8) Process the result.
- 9) 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) Set EDX with the user return address from the stack.
- 8) Set ECX with the value of the user EBP from the stack.
- 9) Return to user mode: *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 **assembler** 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* to know the type and the number of parameters of this assembler 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

3.10.1 Complete Zeos Code

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

- Implement the macro **RESTORE_ALL**.
- Implement the macro **EOI**.
- Implement the keyboard management.
 - Implement the keyboard service routine.
 - Implement the keyboard handler.
 - Initialize the IDT with the keyboard handler
 - Enable the interrupt.
- Implement the *write* system call.
 - 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.

3.10.2 Clock management

The clock management will display the seconds that have elapsed since the boot process. Figure 16 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.c**).

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.

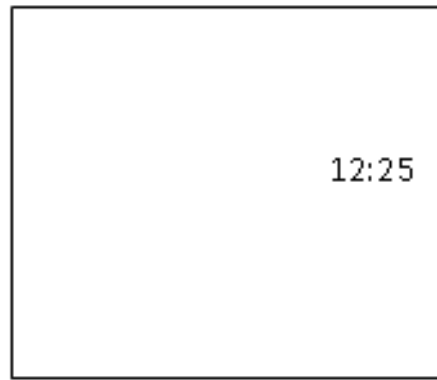


Fig. 16. Clock interrupt prints a clock with the minutes and seconds elapsed.

Writing the 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();
```

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 assembler 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.

Testing

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.

3.10.3 *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 gettime();
```

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.

4 BASIC PROCESS MANAGEMENT

In this section we describe the code and necessary data structures to define and manage processes in ZeOS.

Take into account that the code related to the process management is key to achieve a good system performance. This means that the data structures have to take up minimum space and that the process management algorithms must be highly efficient.

As you know, a process has its own address space. A running process in user mode accesses its user stack, user data and user code. When running in system mode, it accesses the kernel data and code and uses its own system stack.

The process management implies knowledge of memory management. Here you will be given some basic concepts. A lot of work has already been done, but **you will have to understand the code provided to you**.

Chapters 3 and 11 of *Understanding the Linux Kernel* contain information about process management. You can also find useful information about the memory organization in Chapter 2.

This section is the most complex so you are advised to read the following sections carefully and to understand all the concepts explained.

4.1 Prior concepts

- Process. Data related to the process management: PCB (task_struct), lists, etc.
- Process address space. Difference between logical and physical addresses. Paging.
- Context switch. Scheduling. Scheduling policies.

4.2 Definition of data structures

One of the goals of this section is to describe the data structures needed to manage processes in ZeOS. In particular:

- Process Control Block (PCB) implemented with the task_struct structure in sched.h.
- A vector of task_structs to hold the information of several processes.
- The system stack integrated with the task_struct of every process.
- A free queue to link all the tasks that are available for new processes (to get a quick access to them).
- A ready queue to link all the tasks that are candidate to use the CPU which is already busy.

4.2.1 Process data structures

PCB structure

The PCB should contain all necessary information needed to manage the processes in the system. A basic definition of the task_struct is contained in the sched.h file:

```
struct task_struct {
    int PID;                                /* Process ID */
    page_table_entry * dir_pages_baseAddr; /* directory base address */
};
```


This structure has initially only two fields: an integer that will serve as the process identifier (PID), and a pointer to the directory pages of the processes (the directory pages points to the page table of the processes, see section 4.3 for more details). More fields will be added as you advance in the project.

For easy programming, rather than using two different structures: one for the `task_struct` and one for the stack for each process, both data structures will be overlapped in memory. To do so, we use the **union type of C**, which is a special kind of data. In C, if you declare a **union of three fields (a, b and c)**, the compiler reserves memory for the biggest field instead of the sum of their sizes.

To overlap the `task_struct` and the system stack, the `task_union` has been declared. The `task_union` shares the memory space between the process descriptor (`task_struct`) and the system stack of the process. Its declaration is:

```
union task_union {
    struct task_struct task;
    unsigned long stack[KERNEL_STACK_SIZE];
};
```

And you can see the result in Figure 17, where both structures have been depicted sharing the same memory area, a region of 4096bytes.

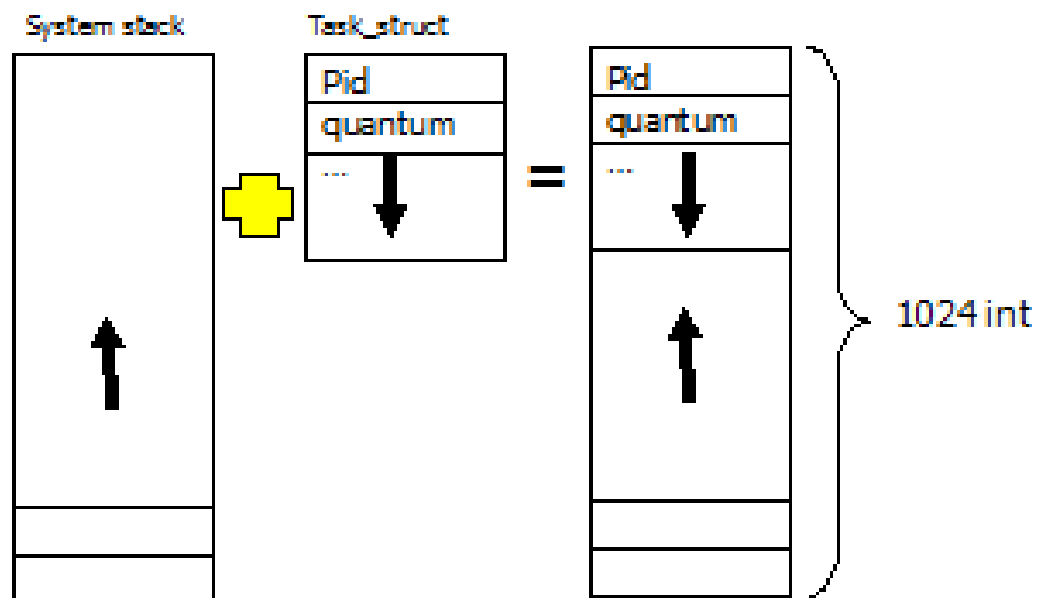


Fig. 17. The system stack and `task_struct` use the same page memory(4k)

The operating system will allow to create and destroy processes, meaning that a bunch of this structures will be needed. To store these processes in the system we will use a fixed size array²⁷. The declaration of the task array with the `task_union` is also provided:

```
union task_union task[NR_TASKS]
```

²⁷. A better option would be to create these `task_struct` structures dynamically, using some kind of dynamic memory allocation but we do not have this feature yet.

```
__attribute__((__section__(".data.task")));
```

The array is tagged with an attribute section to locate this array into the memory section named *data.task*. Figure 18 shows the placement of this task array in memory.

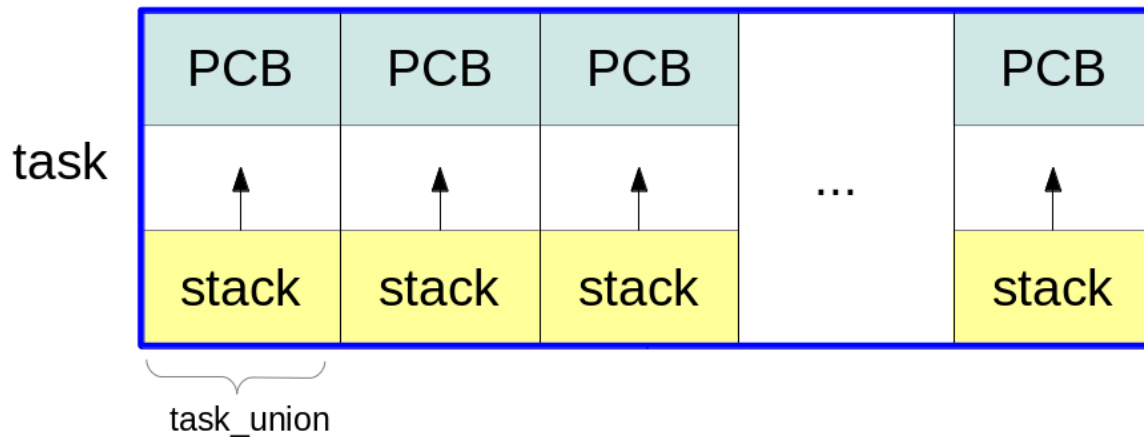


Fig. 18. Memory placement of the task array

The process allocated to entry 0 will be the idle process, referred to as process 0, idle process, or task 0. This process cannot be destroyed.

Free queue

Finally, in order to get a quick access to the task_structs not used in the task array a queue of the available PCBs is desired (freequeue). This queue must be initialized during the operating system initialization to link all the available tasks.

This queue is implemented using the `struct list_head` structure provided in the `list.h` file. In fact, this structure is the one used in Linux to implement whatever all the system queues.

The steps to include the free queue are:

- 1) **Declare and initialize** a variable named `freequeue` of type `struct list_head`.
- 2) Modify the declaration of the `task_struct` structure. Add one field named `list` that you will use to enqueue the structure into a queue.
- 3) Add all `task_structs` to this queue as, at this moment, all of them are available.

Due to the fact that the list functions are generic, they use a unique type: the `struct list_head`. So you will need a function to find the memory address of a `task_struct` given a particular `list_head` struct address. There is one function named `list_head_to_task_struct` which returns a pointer to a `task_struct` given a pointer to a `list_head` struct. Its header is:

```
struct task_struct *list_head_to_task_struct(struct list_head *l);
```

Ready queue

Processes that are candidate to use the CPU but it is currently busy, remain queued in the ready queue. This queue holds the `task_struct` of all the process that are ready to start or continue the execution.

The steps to include the ready queue are:

- 1) **Declare and initialize** a variable named `readyqueue` of type `struct list_head`.

This queue must be initialized during system initialization and it is empty at the beginning.

4.2.2 Process identification

Once you have the process table, an important issue is how to identify a process. When entering the system (through a system call or a clock interrupt, for example) you need to know which process is on execution in order to access its data structures. **Where can you obtain this information? From the system stack pointer (remember that the `task_struct` of a process and its system stack share the same memory page).**

When entering the kernel, you know that:

- The system stack and the `task_struct` are physically in the same space
- The stack size is fixed (its size is 4KB).
- The `task_struct` is located at the beginning of the page that it occupies.
- The `esp` and `ss` registers are changed by the hardware automatically when changing from user mode to system mode.

Thus, if you apply a bitmask to the `esp` register, setting the latest `X` bits to 0 (where `X` is the number of bits used by the stack size), you will have the address of the beginning of the `task_struct`.

A function (or macro) named "current" that returns the address of the current `task_struct`, based on this idea, is provided. This macro gives the pointer to the task that is currently running.

```
struct task_struct * current()
```

4.3 Memory management

This section introduces some concepts and data structures related to memory management, necessary for process management.

In ZeOS, you are provided with the segmentation already initialized (and fixed) and the paging initialized for the initial process. All processes executing on ZeOS access the same logical pages and a logical-to-physical page translation is needed for each process.

In order to understand the creation of the processes, you need some knowledge about the memory organization in the 80x86 architectures.

Two types of ZeOS addresses should be differentiated: logical (or virtual) and physical.

- Logical address²⁸. A 32-bit unsigned integer that represents up to 4GB of memory. The address range is from 0x00000000 to 0xffffffff. All the addresses generated by the cpu are logical addresses and translated to physical addresses by the MMU.
- Physical address. They are represented by 32-bit unsigned integers. Used to address memory from the chip.

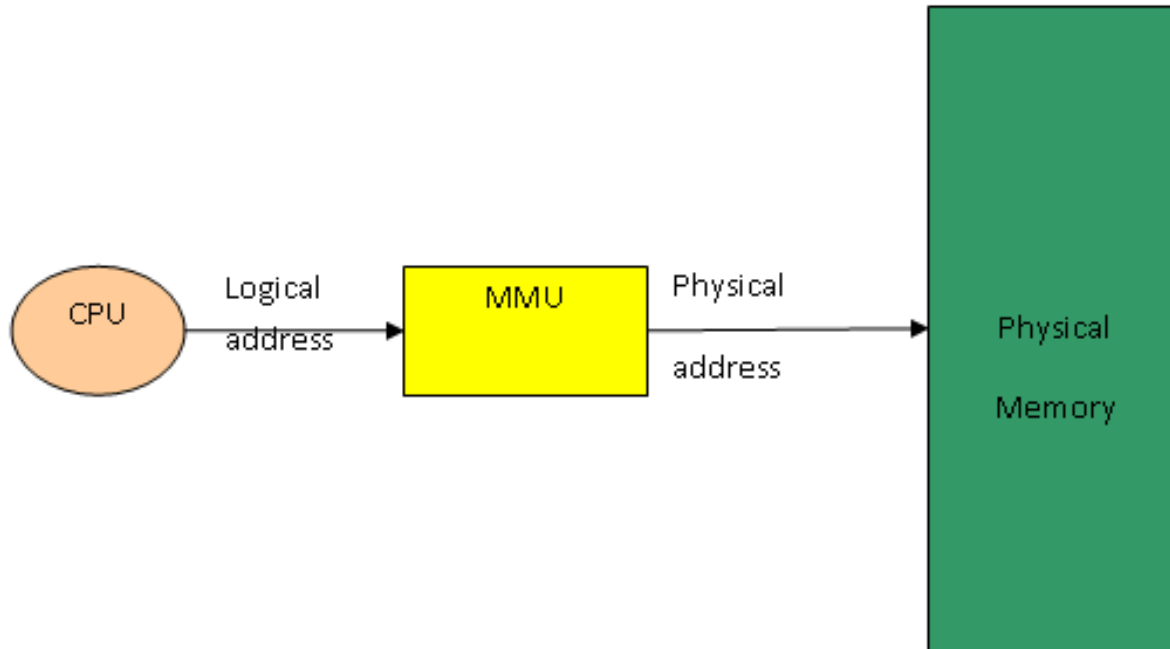


Fig. 19. Translation from logical addresses to physical addresses

Figure 19 shows the relationship between the different addresses. A brief description of segmentation and paging concepts will now be given.

4.3.1 MMU: Paging mechanism

Paging translates logical addresses into physical addresses. The memory is arranged in page frames of equal size. The data structure that translates linear addresses into physical addresses is the page table.

To activate the paging mechanism in the 80x86 architectures, it is necessary **to activate the PG bit of the cr0 register**. PG=0 means that the linear addresses are the same as the physical addresses. This initialization is already done in the code provided in ZeOS.

4.3.2 The directory and page table

The addressing mode in ZeOS has to use the two indexing levels offered by the 80x86 paging architecture: *page directory* and *page table*. Each page directory contains the base addresses of the page tables defined in the system for one process. The page tables contain the actual translations

²⁸. A difference is established between logical and linear in "Understanding the Linux Kernel". What we refer to as "logical" here is called "linear" in the book.

from logical to physical pages of the processes. `cr3` is a special register that keeps the physical base address²⁹ of the page directory used by the MMU to do the current translations.

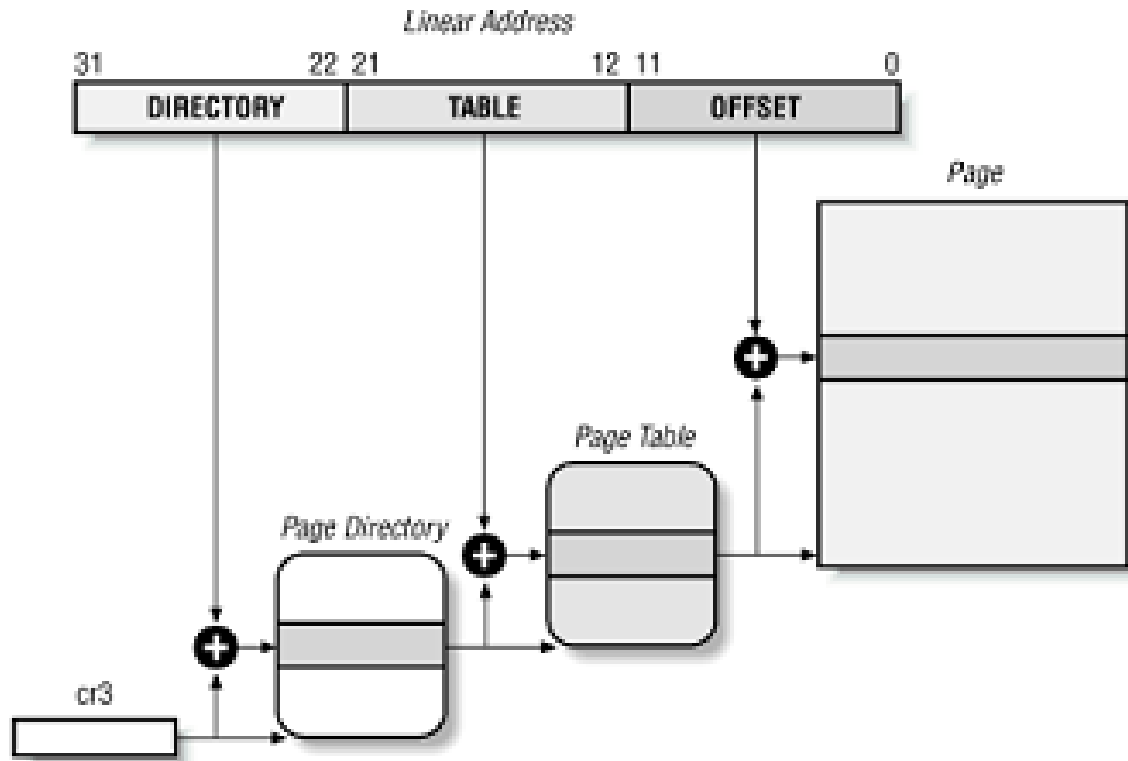


Fig. 20. Paging in the 80x86 architecture

Figure 20 shows the basic address translation mechanism based on paging used in ZeOS. In the Intel architecture, the 10 most significant bits (bits 31-22) of a 32-bit address define the entry to the *page directory*. The 10 bits in the middle (bits 21-12) define the *logical page* number and indexes the page table. The 12 least significant bits (bits 11-0) define the *offset* within the page (up to 4KB). Therefore, to obtain the logical page number for a given address, it is only necessary to shift it to the right: $\text{page_number} = (\text{address} \gg 12)$.

The purpose of the two indexing levels mechanism is to allocate space for the page tables as it is needed, this is as the address space of a process grows. In ZeOS, the address space of all processes is small and only one page table per process is needed. Therefore each page directory has only one valid entry (entry 0).

Each process has its own page directory and page table, and the operating system is the responsible to load the `cr3` register with the base address of the page directory for the process that is going to use the cpu. For this reason, ZeOS keeps the base address of the directory pages of each process in its PCB (field `dir_pages_baseAddr` in the `task_struct` structure).

29. Addresses for all kernel code and data, included page directories, are absolute. This means that logical addresses are the same than physical addresses.

Variables *dir_pages* and *pagusr_table* are two fixed sized vectors that hold the page directories and page tables respectively for each process in ZeOS (see file mm.c). These vectors have NR_TASKS entries, and each of these entries is linked to a *task_struct* in the *tasks* vector, through the *dir_pages_baseAddr* field.

The initialization of all the page directories is given as part of the ZeOS code (see function *init_dir_pages* in file mm.c). This initialization consists on setting the entry 0 of each page directory to point to its corresponding page table.

The page directory and the page table entries have the same structure. Each entry has the following flags (check the ZeOS code):

```
typedef union
{
    unsigned int entry;
    struct {
        unsigned int present : 1;
        unsigned int rw      : 1;
        unsigned int user    : 1;
        unsigned int write_t : 1;
        unsigned int cache_d : 1;
        unsigned int accessed : 1;
        unsigned int dirty   : 1;
        unsigned int ps_pat  : 1;
        unsigned int global  : 1;
        unsigned int avail   : 3;
        unsigned int pbase_addr : 20;
    } bits;
} page_table_entry;
```

Each entry has been defined as an union to initialize it easily. Setting the field *entry* to 0, sets the whole structure to 0. A new routine to enable this operation has been defined:

```
void del_ss_pag(page_table_entry* PT, unsigned logical_page);
```

The *pbase_addr* field contains the page frame number that will be used for the current logical page. Therefore, if this structure is an entry of a page table³⁰, it will translate from a **logical page number to a physical page number**. Remember that the content of the *pbase_addr* field is the **physical page number, not the address**. To change the association between a logical page and a physical frame, you can use this function (mm.c):

```
void set_ss_pag(page_table_entry* PT, unsigned logical_page, unsigned
physical_frame);
```

4.3.3 Translation Lookaside Buffer (TLB)

The TLB is a table that the architecture uses to optimize the memory access. The page table entries used by the current process are stored in this table. The TLB is like a cache of the page table entries. This helps us to save one access to the directory when translating the linear addresses

30. Instead of an entry of a page directory.

into physical ones. Moreover, the TLB access is very fast. **Bear in mind that the TLB is not synchronized automatically with the page table.** Therefore, when the page table is modified, the associated TLB entries can become incorrect, so these TLB entries must be invalidated. In particular, every time a page table entry is deleted or modified, it is necessary to invalidate the TLB. This action is called TLB flush. In the 80x86 paging architecture, the hardware flushes the TLB each time the value of the cr3 register changes. **To invalidate the TLB, rewrite the cr3 register using the routine `set_cr3(page_table_entry* dir)` (see file `mm.c`).**

4.3.4 Specific organization for ZeOS

Logical space

The logical space of a process is the memory space addressable for user programs. It is composed of segments and pages. The ZeOS segmentation has been reduced to a minimum, like in Linux. If you look at the `bootsect.S` file, you will see how the GDT entries have been initialized and that the segments are defined to start in the address 0 with an infinite size. This means that a segment:offset address will be translated basically to a linear address containing the same value of the offset. To make everything work correctly and due to the lack of a loader, the linker will generate addresses for user programs starting at the `0x100000` (`L_USER_START` in `mm_address.h`). In order to have a global understanding of memory management, it is necessary to understand the organization of segments. However, it is not necessary to modify the one provided.

Logical space in ZeOS

The linear address space is composed of N code pages and M data+stack pages, both consecutive in memory.

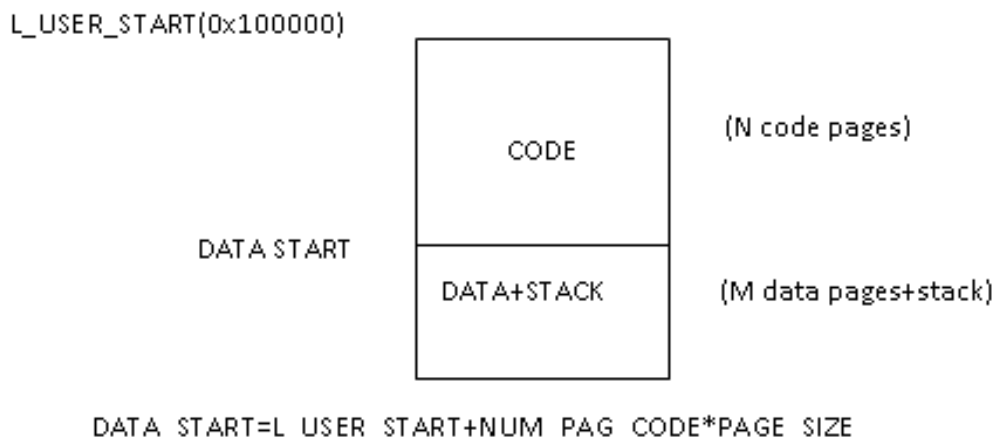


Fig. 21. Linear address space

Figure 21 shows the user addresses of a process. The left side of the figure shows the logical addresses of any process. The logical space of all processes starts at the address `L_USER_START` and has `NUM_PAGE_CODE` code pages and `NUM_PAG_DATA` pages of data.

In the case of ZeOS, the first 255 entries of the page table (addresses `0x0` to `0xfffff`) correspond to the system space (see the initialization code in file `mm.c`). Addresses of the system space are absolute, i.e. they do not require translation because logical addresses match physical address.

Each entry represents 4KB (it is the page size). $L_USER_START \gg 12$ is the first logical page of the user code (you must define a constant to access this page in the file `mm_address.h`). Therefore, the entire process has $255 + NUM_PAGE_CODE + NUM_PAGE_DATA$ valid entries in the page table: 0-255 for the system, only accessible when in kernel mode, NUM_PAGE_CODE pages for the user code and NUM_PAGE_DATA pages for the data and user stack. In the initial ZeOS configuration, $NUM_PAGE_CODE=8$ and $NUM_PAGE_DATA=20$ (see file `mm_address.h`).

Both logical and physical pages for system and user code are shared. This means that the logical entries in the page table of all processes point to the same physical pages.

The pages for user data+stack are private and therefore the translation from logical addresses to physical addresses (or frame) will be different for each process.

For each new process, it is necessary to initialize its whole page table with the suitable translation for each logical page.

As few pages will be used, a single entry of each page directory will be used (`ENTRY_DIR_PAGES`). This entry has the address of the corresponding page table. And register `cr3` points to the base address of the current page directory.

For each context switch it is necessary to change the value of the `cr3` register to point to the process that it is going to use the cpu.

Remember that the hardware uses the information in the page table to translate all the accesses to memory: it checks that the type of access is enable for the target logical address and translate the address to complete the access. If the hardware is not able to complete the access it generates an exception.

In order to avoid page faults inside the operating system, ZeOS has to check that all the addresses passed by the user as parameters are valid. For this purpose, a function named *access_ok* has been provided. This function, given a user memory address, a block size and a type of access, checks in the page table of the process if the access is valid.

```
int access_ok(int type, const void *addr, unsigned long size)
type READ or WRITE
addr user address
size block size to check
returns 1 if correct and 0 otherwise
```

Physical space

As mentioned above, **the logical space is consecutive but the physical space is not**. The physical memory will be organized as follows: the code pages will be shared by all the processes and they will start at `PH_USER_START` (defined in `segment.h`). Starting at this address, the system will place memory pages for user data and user stacks of created processes (see Figure 22).

Notice that the only difference between the various processes will be the entries of the page table corresponding to the user data and user stack.

The physical pages or frames are computed from physical addresses. For example, the physical pages for the code of process `P0` will start at the frame $PH_USER_START \gg 12$. A macro name `PH_PAGE` to compute the page number starting from a memory address has been provided:

```
#define PH_PAGE(x) (x >> 12)
```

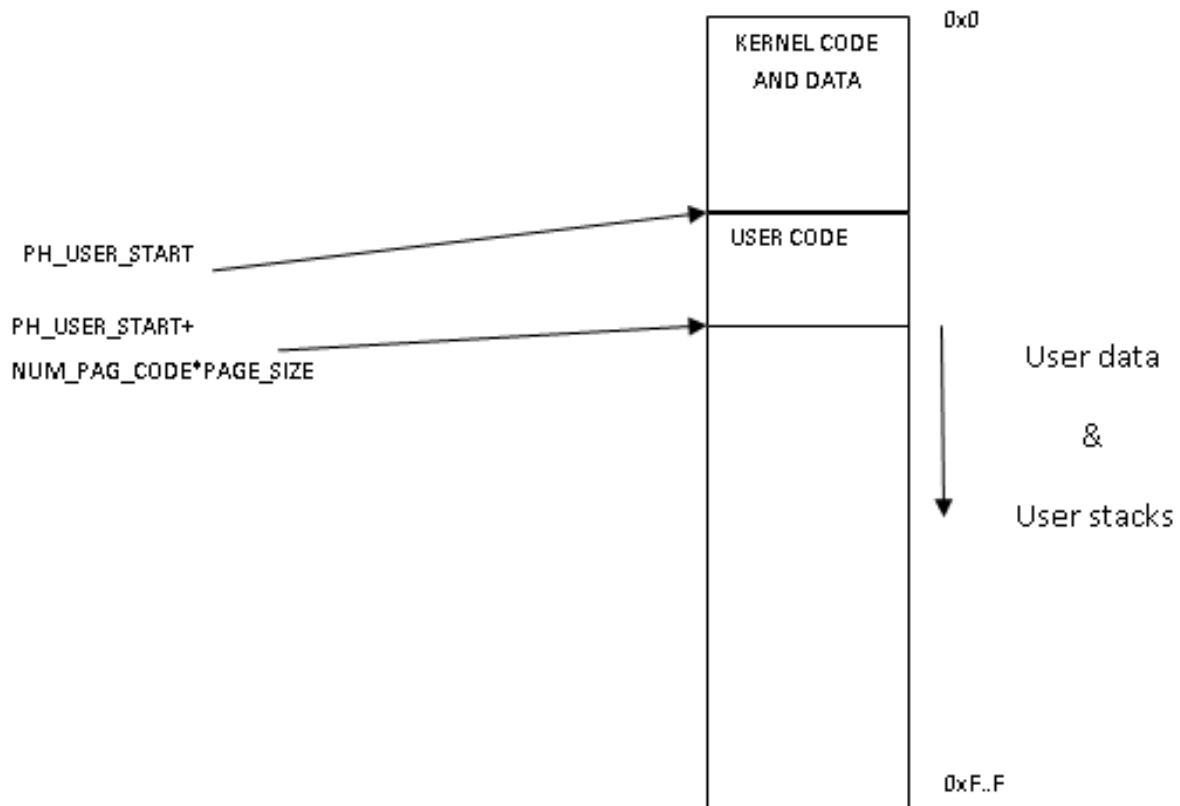



Fig. 22. Physical memory in Zeos

Every time a process is created, it will be necessary to look for available frames in the physical memory for mapping data and stack to the process. And every time a process ends an execution, it will mark as available the data and stack frames that were assigned to the process.

The `phys_mem` table (`mm.h` file) has been defined to have information in the system about busy frames (assigned to a logical page of a process) and free frames. It contains an entry for each frame with its status. The structure is initialized as a part of the `init_mm` routine (in `mm.c`) and it is based on marking all system frames and the frames used by the user code as busy (see Figure 23).

To complete the `phys_mem` management interface, you are provided with the `alloc_frame` function (to allocate one frame) and the `free_frame` (to deallocate one frame) functions in the `mm.c` file. The interface of these two functions is described below.

```
int alloc_frame(void)
```

The `alloc_frame` function searches one available frame. If it finds one, then it is marked as busy in the `phys_mem` table and returned. If there are no free frames, it returns -1.

```
void free_frame(unsigned int frame)
```

The `free_frame` function marks the frame passed as a parameter as free.

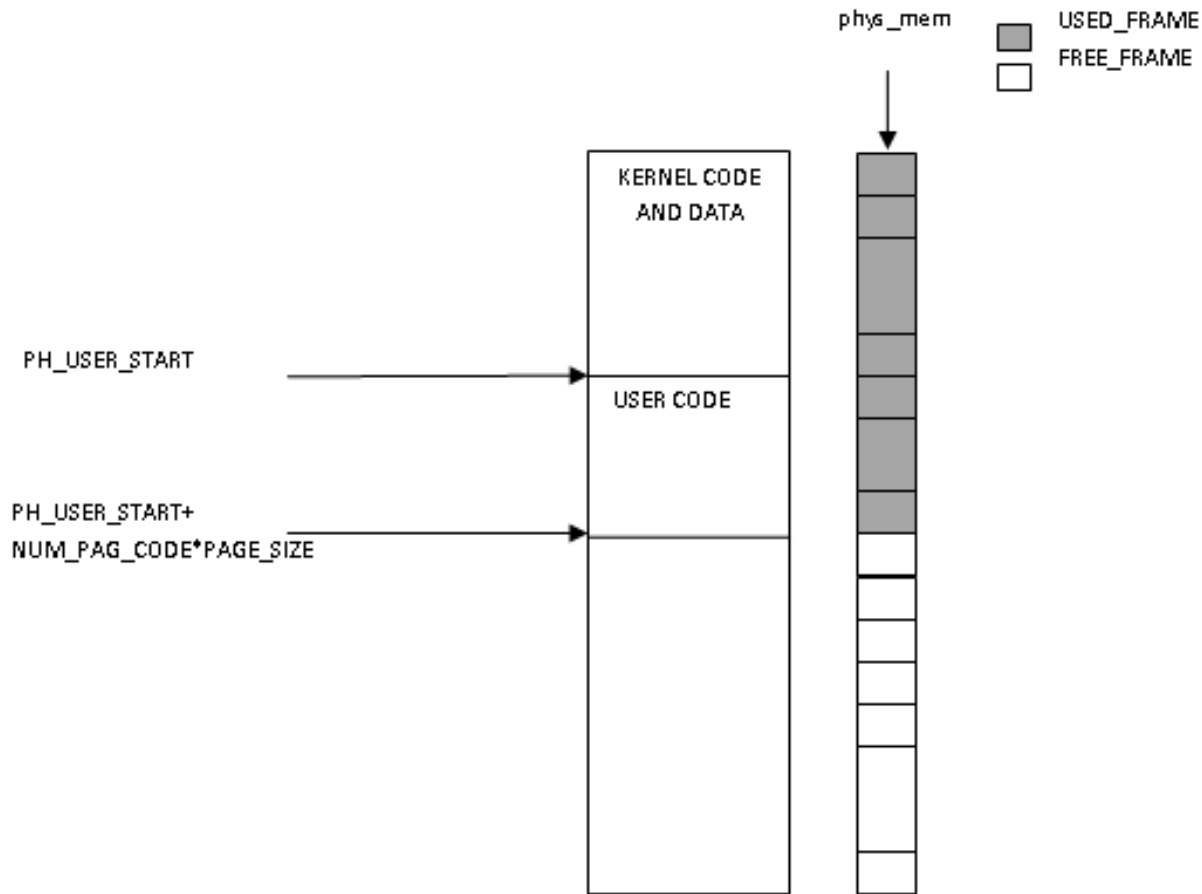


Fig. 23. Initializing physical memory and `phys_mem` table

4.4 Initial processes initialization

There are two special processes in ZeOS: the idle process and the init process. Both processes are created and initialized during the initialization code of the system (see file `system.c`).

4.4.1 Idle process

The **idle process** is a kernel process that never stops and that always execute the funtion `cpu_idle`, which executes the assembler instruction `sti` and implements an empty endless loop (see file `sched.c`).

The assembler instruction `sti` is required because in ZeOS, interrupts are disabled while executing in kernel mode (disabling interrupts is the default behavior when an interrupt happens in the 80x86 architecture). As the idle process executes in kernel mode, it is necessary to explicitly enable interrupts when the idle process starts using the `cpu`, in order to be able to interrupt it and to give the `cpu` to any user process that becomes ready to use it.

The PID of this process is 0, always executes on kernel mode and only should use the CPU if there are not any other user process to execute (its purpose is just to keep the CPU always busy).

The main steps to initialize the idle process have to be implemented in the `init_idle` function and are the following:

- 1) Get an available task_union from the freequeue to contain the characteristics of this process
- 2) Assign PID 0 to the process.
- 3) Initialize field *dir_pages_baseAaddr* with a new directory to store the process address space using the *allocate_DIR* routine.
- 4) Initialize an execution context for the process to restore it when it gets assigned the cpu (see section 4.5) and executes *cpu_idle*.
- 5) Define a global variable *idle_task*

```
struct task_struct * idle_task;
```

- 6) Initialize the global variable *idle_task*, which will help to get easily the *task_struct* of the idle process.

The first time that the idle process will use the cpu will be due to a cpu scheduling decision, when there are not more candidates to use the cpu. Thus, the context switch routine will be in charge of putting this process on execution. This means that we need to initialize the context of the idle process as the context switch routine requires to restore the context of any process. This context switch routine restores the execution of any process in the same state that it had before invoking it (see subsection 4.5 for more details). This is achieved by undoing the dynamic link in the stack and then return using the code address in top of the stack (see figure 24). This means that we need to:

- 1) **Store in the stack of the idle process the address of the code that it will execute (address of the *cpu_idle* function).**
- 2) **Store in the stack the initial value that we want to assign to register *ebp* when undoing the dynamic link** (it can be 0),
- 3) Finally, we need to **keep (in a field of its *task_struct*) the position of the stack where we have stored the initial value for the *ebp* register.** This value will be loaded in the *esp* register when undoing the dynamic link.

Notice that as part of this initialization it is not necessary to modify the page table of this process. The reason is that this process is a kernel process and only needs to use those pages that contain kernel code and kernel data which are already initialized during the general memory initialization in the *init_mm* function (see file *mm.c*). This function allocates all the physical pages required for the kernel pages (both code and data pages) and initializes all the page tables in the system with the translation for these kernel pages, which are common for all the processes (see functions *init_frames* and *init_table_pages* in file *mm.c*).

4.4.2 Init process

The **init process** is the first user process to be executed after booting the OS. Its PID is 1 and it executes the user code. Since the init process is the first one executed, it becomes the parent for the rest of processes in the system. The code for this process is implemented in the *user.c* file.

The steps to initialize the init process have to be implemented in the *init_task1* function which is called from the function *main* in *system.c*. Those steps are:

- 1) Assign PID 1 to the process
- 2) Initialize field *dir_pages_baseAaddr* with a new directory to store the process address space using the *allocate_DIR* routine.

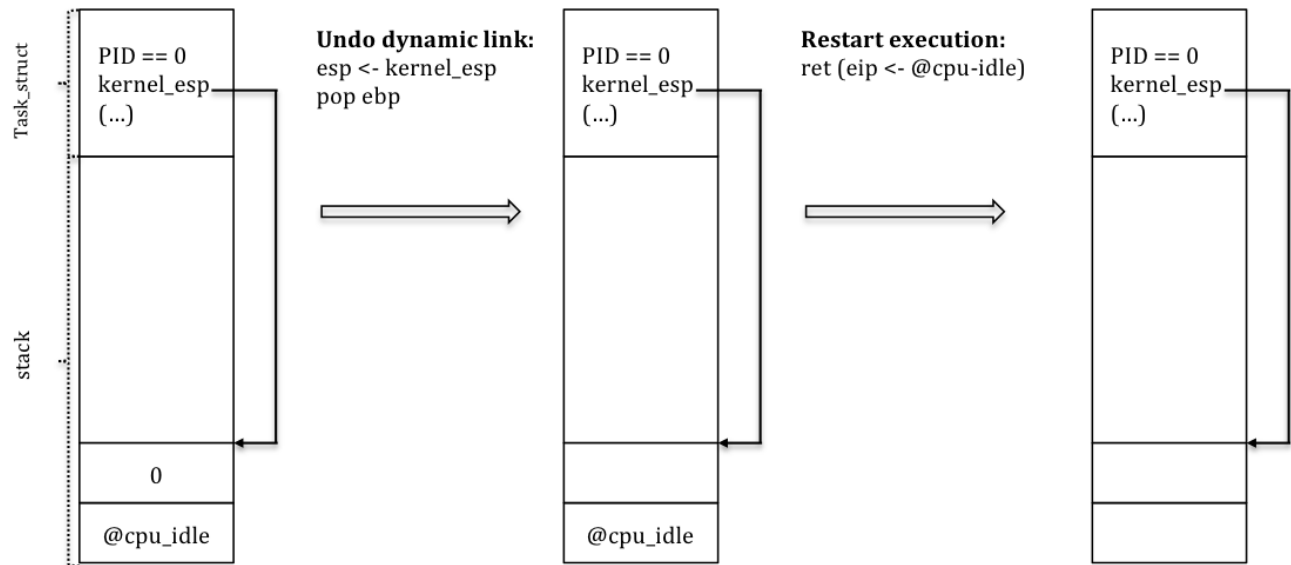


Fig. 24. Idle process: putting it on execution

- 3) Complete the initialization of its address space, by using the function `set_user_pages` (see file `mm.c`). This function allocates physical pages to hold the user address space (both code and data pages) and adds to the page table the logical-to-physical translation for these pages. Remind that the region that supports the kernel address space is already configure for all the possible processes by the function `init_mm`.
- 4) Update the TSS to make it point to the new `_task` system stack. In case you use `sysenter` you must modify also the MSR number `0x175`.
- 5) Set its page directory as the current page directory in the system, by using the `set_cr3` function (see file `mm.c`).

This process is the first process that will use the cpu after the system initialization. This means that the context switch routine is not involved with the first execution of this process. It is the function `return_gate` which prepares the stack of the process to enable the execution of the user code (see file `hardware.c`). At the end of `return_gate`, the `lret` assembler instruction is executed to downgrade the privilege level to user level and to execute the main function in the `user.c` file.

4.4.3 Zeos implementation details

As part of the initialization code it is necessary to initialize both processes: the idle and the init. In file `sched.c` you have defined two empty functions for this purpose: `init_idle` and `init_task1`. These functions are invoked from the system `main` function (see file `system.c`), and you have to complete them with the necessary steps to initialize both processes. In addition, **you have to delete the sentence that invoke the function `monoprocess_init_addr_space`**. This function initializes the address space of the monoprocess version of ZeOS and it is not needed for the multitasking version that you are building now.

4.5 Process switch

Zeos provides a routine for making the context switch between two processes. This context switch is performed in kernel mode.

The context switch consists of exchanging the process that is being executed with another process. It changes the user address space (changing the page table), changes the kernel stack (to restore the new context), and restores the new hardware context. This routine will be named *task_switch*.

Its header is:

```
void task_switch(union task_union *new)
new: pointer to the task_union of the process that will be executed
```

This routine 1) saves the registers ESI, EDI and EBX³¹, 2) calls the *inner_task_switch* routine, and 3) restores the previously saved registers.

The *inner_task_switch* routine has the same header as the previous *task_switch*, and it restores the execution of the *new* process in the same state that it had before invoking this routine. This is achieved by changing to the new kernel stack, undoing the dynamic link in the stack (usually created by the C compiler when a function is called) and then continue the execution at the code address in the top of the stack. Figure 25 shows the stages of the stack of the current process during the invocation of the context switch. The last picture is the expected state of any process stack to be restored.

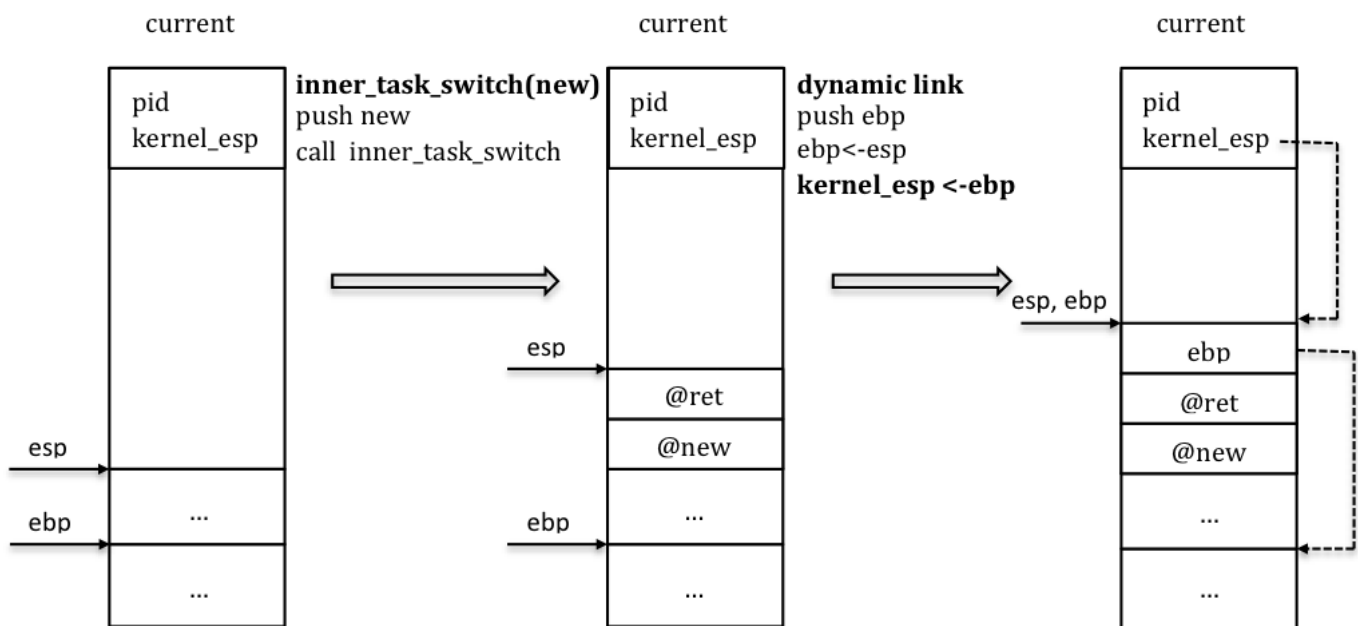


Fig. 25. Stack stages during the context switch invocation

The following steps are needed to perform a context switch:

31. Usually the compiler is smart enough to detect that these registers are modified inside the function code and it saves them automatically, but, in this case, the compiler is not aware of the whole code of the function (due to the trickeries that you will see soon) and potentially these registers may be modified, meaning that these registers would have a different value when returning from this function. The easy solution is to save and restore them manually.

- 1) Update the pointer to the system stack to point to the stack of `new_task`. This step depends on the implemented mechanism to enter the system. In the case that the `int` assembler instruction is used to invoke the system code, `TSS.esp0` must be modified to make it point to the stack of `new_task`. If the system code is invoked using `sysenter`, MSR number 0x175 must be also modified.
- 2) Change the user address space by updating the current page directory: use the `set_cr3` function to set the `cr3` register to point to the page directory of the `new_task`.
- 3) Store the current value of the `EBP` register in the `PCB`. `EBP` has the address of the current system stack where the `inner_task_switch` routine begins (the dynamic link).
- 4) Change the current system stack by setting `ESP` register to point to the stored value in the `new PCB`.
- 5) Restore the `EBP` register from the stack.
- 6) Return to the routine that called this one using the instruction `RET` (usually `task_switch`, but...).

The context switch is used when the scheduling policy decides that a new process must be executed, or when a process blocks.

4.5.1 Initial testing

You may use the `idle` and `init` processes to check that it works (for example, executing the `init` process at the beginning and after some ticks (or after a keypress) make a `task_switch` to the `idle`).

4.6 Process creation and identification

In this part, two system calls related to process management will be described in ZeOS: **`getpid`** and **`fork`**. `getpid` returns the `pid` of the process that calls it. `fork` creates a new process, a copy of the one that calls it (referred to as parent), inserts it into the processes list, ready to execute, and returns its `pid`.

4.6.1 getpid system call implementation

The header of the call is:

```
int getpid(void)
```

It must return the `pid` of the process that called it. The identification of the `getpid` call is 20.

Since this is a very simple system call, its implementation is straightforward.

The wrapper routine moves to `eax` the identifier of the `getpid` syscall (20), traps the kernel and returns the corresponding `PID` also in the `eax` register. The service routine returns the `PID` in the `task_struct` of the current process.

4.6.2 fork system call implementation

The header of the call is:

```
int fork(void)
```

It returns a different value depending on whether it is the parent or the child. If it is the parent, it returns the pid of the created process; if it is the child, it returns a 0. If an error occurs it returns a -1.

Regarding the memory initialization, it is necessary to allocate free physical pages to hold the user data+stack region of the child process: system data and code are shared by all processes and since user code is read-only it is not necessary to replicate it. The data+stack region is inherited from the parent and is private thereafter. This means that the new allocated pages will be initialized as a copy of the data+stack region of the parent.

The implementation of the fork system calls carries out the following steps:

- 1) Implement the wrapper of the system call. The identifier of the system call is 2.
- 2) Implement the `sys_fork` service routine. It must:
 - a) Get a free `task_struct` for the process. If there is no space for a new process, an error will be returned.
 - b) Inherit system data: copy the parent's `task_union` to the child. Determine whether it is necessary to modify the page table of the parent to access the child's system data. The `copy_data` function can be used to copy.
 - c) Initialize field `dir_pages_baseAddr` with a new directory to store the process address space using the `allocate_DIR` routine.
 - d) Search physical pages in which to map logical pages for data+stack of the child process (using the `alloc_frames` function). If there is no enough free pages, an error will be return.
 - e) Inherit user data:
 - i) Create new address space: Access page table of the child process through the directory field in the `task_struct` to initialize it (`get_PT` routine can be used):
 - A) Page table entries for the system code and data and for the user code can be a copy of the page table entries of the parent process (they will be shared)
 - B) Page table entries for the user data+stack have to point to new allocated pages which hold this region
 - ii) Copy the user data+stack pages from the parent process to the child process. The child's physical pages cannot be directly accessed because they are not mapped in the parent's page table. In addition, they cannot be mapped directly because the logical parent process pages are the same. They must therefore be mapped in new entries of the page table temporally (only for the copy). Thus, both pages can be accessed simultaneously as follows:
 - A) Use temporal free entries on the page table of the parent. Use the `set_ss_pag` and `del_ss_pag` functions.
 - B) Copy data+stack pages.
 - C) Free temporal entries in the page table and flush the TLB to really disable the parent process to access the child pages.
 - f) Assign a new PID to the process. The PID must be different from its position in the `task_array` table.
 - g) Initialize the fields of the `task_struct` that are not common to the child.
 - i) Think about the register or registers that will not be common in the returning of the child process and modify its content in the system stack so that each one receive its values when the context is restored.

- h) Prepare the child stack with a content that emulates the result of a call to `task_switch`. This process will be executed at some point by issuing a call to `task_switch`. Therefore the child stack must have the same content as the `task_switch` expects to find, so it will be able to restore its context in a known position. The stack of this new process must be forged so it can be restored at some point in the future by a `task_switch`. In fact this new process has to a) restore its hardware context and b) continue the execution of the user process, so you must create a routine *ret_from_fork* which does exactly this. And use it as the restore point like in the idle process initialization 4.4.
- i) Insert the new process into the ready list: `readyqueue`. This list will contain all processes that are ready to execute but there is no processor to run them.
- j) Return the pid of the child process.

4.7 Process scheduling

In order to execute the different processes queued in the ready queue, a policy must be defined to select the next process. In this section, you will add a process scheduler that uses a round robin policy. However, the code of this scheduler must be generic enough to replace the round robin policy with a different one. For this reason we define an interface for the main scheduling functions.

4.7.1 Main scheduling functions

Following we describe the interface that you have to implement to provide ZeOS with a scheduling policy.

- Function to update the relevant information to take scheduling decisions. In the case of the round robin policy it should update the number of ticks that the process has executed since it got assigned the cpu.

```
void update_sched_data_rr (void)
```

- Function to decide if it is necessary to change the current process.

```
int needs_sched_rr (void)
returns: 1 if it is necessary to change the current process and 0
otherwise
```

- Function to update the current state of a process to a new state. This function deletes the process from its current queue (state) and inserts it into a new queue (for example, the free queue or the ready queue). If the current state of the process is *running*, then there is no need to delete it from any queue. The parameters of this function are the *task_struct* of the process and the queue according to the new state of the process. If the new state of the process is *running*, then the queue parameter should be `NULL`.

```
void update_process_state_rr (struct task_struct *t, struct list_head *dst_queue)
```

- Function to select the next process to execute, to extract it from the ready queue and to invoke the context switch process. This function should always be executed after updating the state of the current process (after calling function *update_process_state_rr*).


```
void sched_next_rr (void)
```

You have to:

- 1) Add the necessary fields to the `task_struct` to implement the scheduler.
- 2) Create a Round Robin scheduling policy. To do this, you should implement the previous interface and you should add the code necessary to invoke those functions.

4.7.2 Round robin policy implementation

The round robin policy consists in executing each process during a specific number of ticks (its *quantum*), doing a `task_switch` when the quantum finishes. This quantum can be different for each process and each process inherits it from its parent. Ticks are updated in the clock interrupt; the scheduling policy will then be invoked there to check if a new process must be executed.

Adding this policy requires implementing a new function, named *schedule*, and calling it whenever a scheduling decision is needed. The steps that this function has to perform are:

- Control how many ticks the current process has spent on the CPU. Use a global variable that is decreased with every tick.
- Decide if a context switch is required. A context switch is required when the quantum is over and there are some candidate process to use the CPU. Recall that the idle process should use the CPU **ONLY** if there are not any other user process that can advance with the execution.
- If a context switch is required:
 - Update the readyqueue, if current process is not the idle process, by inserting the current process at the end of the readyqueue.
 - Extract the first process of the readyqueue, which will become the current process;
 - And perform a context switch to this selected process. Remember that when a process returns to the execution stage after a context switch, its quantum ticks must be restored.

Additionally, to complete the implementation it is necessary to add a function named *get_quantum* that returns the quantum of the task passed as a parameter and a function *set_quantum* that modifies the quantum of the process.

```
int get_quantum (struct task_struct *t);
void set_quantum (struct task_struct *t, int new_quantum):
```

Finally, check that the round robin policy works correctly by showing the pid of the running process.

4.8 Process destruction

This section describes the steps to destroy a process. This destruction is implemented in the *exit* system call, which destroys the process that calls it. Therefore, it must delete the process from the processes table and make a context switch.

Its header is:

```
void exit(void)
```

The steps to implement the exit system call are:

- 1) Implement the wrapper of the system call. Its identifier is 1.
- 2) Implement the service routine `sys_exit`. In this case, this system call does not return any value.
 - a) Free the data structures and resources of this process (physical memory, `task_struct`, and so). It uses the `free_frame` function to free physical pages.
 - b) Use the scheduler interface to select a new process to be executed and make a context switch.

4.9 Statistical information of processes

The objective of this section is to add statistical information about processes in ZeOS. In particular, for each process, it must keep information about the **time spent in user mode**, the **time spent in the ready queue**, and the **time spent in the blocked state**³². In addition it must also keep information about the **number of context switches**³³ involving the process and the amount of **ticks remaining in the current quantum of the process**.

The `stats.h` file contains a data structure to be used in this new functionality. Remember that the definition of this structure will be used both by the user and by the kernel.

```
struct stats {
    unsigned long user_ticks; /* Total ticks executing user code */
    unsigned long system_ticks; /* Total ticks executing system code */
    unsigned long blocked_ticks; /* Total ticks in the blocked state */
    unsigned long ready_ticks; /* Total ticks in the ready state */
    unsigned long elapsed_total_ticks; /* Ticks since the power on of the machine
    until the beginning of the current state */
    unsigned long total_trans; /* Total transitions ready --> run */
    unsigned long remaining_ticks; /* Remaining ticks to end the quantum */
};
```

Modify the `task_struct` as required to keep the statistical data, implement the code to keep updated this data for each process and implement the new system call to access it.

The already implemented function `get_ticks` provides the elapsed ticks since the power on of the machine, and it can be used to calculate the different times. It has the following interface:

```
unsigned long get_ticks(void)
```

You must use this function each time a process changes the state both: 1) to update the amount of time spent in the state that the process is leaving and 2) to register into the field `elapsed_total_ticks` the initial time for the next state. For example, each time a process enters the system you have to update the `user_ticks` field and to start counting the system time by performing the following operations:

```
current_ticks = get_ticks();
current->user_ticks += current_ticks - current-> elapsed_total_ticks;
current->elapsed_total_ticks = current_ticks;
```

32. Notice that until now we have not implemented any blocking system call and thus, this time will be 0 for all the processes.

33. Or more accurately, the number of times that the process has finished its quantum.

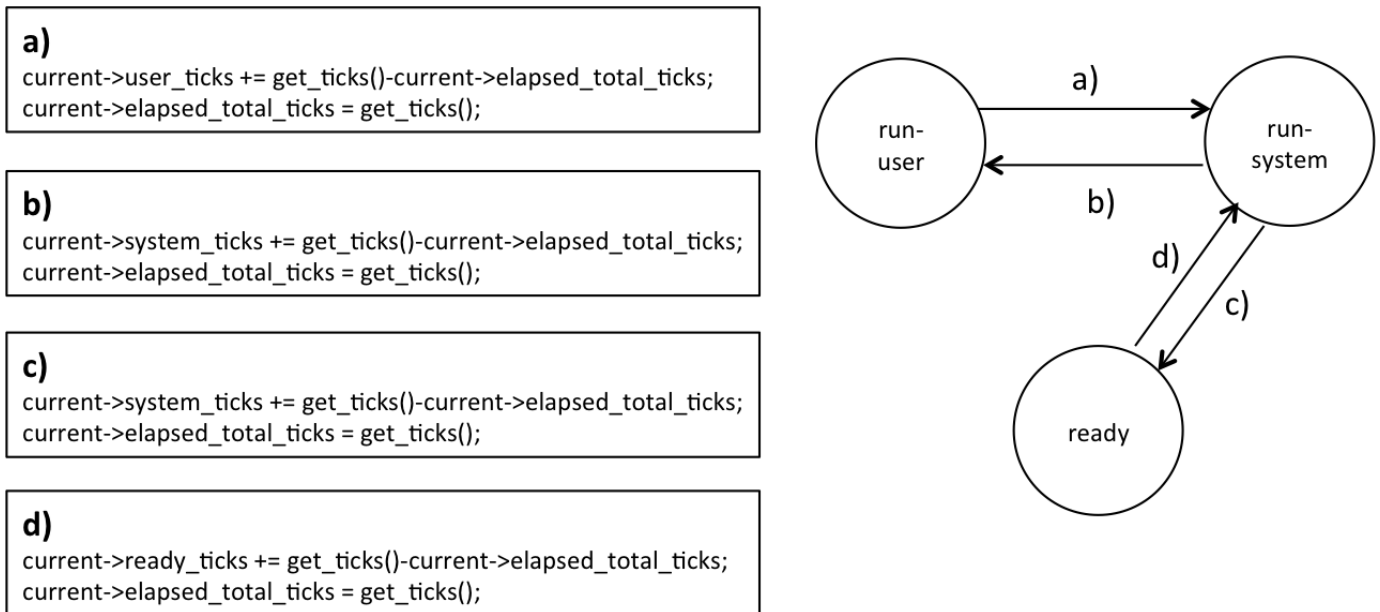


Fig. 26. States to consider when updating the execution time of a process

Figure 26 represents the states and transitions that we are considering at this stage of the project, and the code involved to update the statistical information about the execution time. Think carefully about where to insert each of these lines of code inside the ZeOS implementation.

The identifier of the `get_stats` system call is 35. This system call returns -1 if an error is produced, 0 otherwise.

```

int get_stats(int pid, struct stats *st)
pid: process identifier whose statistics get_stats will get
st: pointer to the user buffer to store the statistics

```

The `pid` parameter contains the identifier of an alive process. The statistics of the process will be returned through the `st` parameter. Be aware that `st` is a pointer to the user address space. Think about error cases and how to deal with them.

4.10 Work to do

This section describes the steps to follow to make ZeOS become multitasking. You will complete the ZeOS code to create/identify/switch/destroy processes and add a new process scheduler:

- Adapt the `task_struct` definition.
- Initialize a free queue.
- Initialize a ready queue.
- Implement the initial processes initialization.
- Implement the `task_switch` function.
- Implement the `inner_task_switch` function.
- Implement the `getpid` system call.
- Implement the `fork` system call.
- Implement process scheduling.
- Implement the `exit` system call.

- Implement the *getstats* system call.

5 BLOCKING OPERATIONS AND PERFORMANCE EVALUATION

The goal of these laboratory sessions is to analyze and to compare the performance of several workloads using different scheduling policies³⁴.

To meet this goal you should adapt your ZeOS code to support different scheduling policies and to enable blocking operations. We provide you with a header file, *schedperf.h*, and with two new libraries, *libschedperf.a* (with system code) and *libuser.a* (with user code), which contain some code that will help you to add the required additional features to ZeOS. You have to modify your Makefile to link your code with these two new libraries. We also provide you with a spreadsheet to fill out with the results from your experiments

5.1 Support to different scheduling policies

An easy methodology to support different scheduling policies consists of the following steps:

- 1) Define a clear interface to implement all scheduling policy code
- 2) Implement a version of this interface for each scheduling policy that will be supported
- 3) Define scheduling interface variables: a set of variables to point to the interface functions of the current scheduling policy.
- 4) Execute all scheduling code by using this set of variables.
- 5) Implement a system call to change the current scheduling policy

5.1.1 What is already done

- In the previous submission you have already defined a scheduling interface and you have implemented a version of this interface for the round robin policy.
- Library *libschedperf.a* contains the declaration of the scheduling interface variables and file *schedperf.h* contains their *extern* definition:

```
extern void (*sched_next) (void);
extern void (*update_process_state) (struct task_struct *,struct list_head*);
extern int (*needs_sched) ();
extern void (*update_sched_data) ();
```

- Implementation of function *init_sched_policy()*, which initializes the scheduling interface variables to point to the FCFS functions. It is necessary to call this function from *init_sched()* function. This function is implemented as follows:

```
void init_sched_policy() {
    sched_next=sched_next_fcfs;
    update_process_state = update_process_state_fcfs;
    needs_sched = needs_sched_fcfs;
    update_sched_data = update_sched_data_fcfs;
}
```

34. We can define workload as the amount of work that a system has to perform. In our context, a set of process that run concurrently and that are the input for the scheduling policy.

- Implementation of FCFS scheduling interface: library *libschedperf.a* contains implementation of functions *sched_next_fcfs*, *update_process_state_fcfs*, *needs_sched_fcfs* and *update_sched_data_fcfs*.
- Implementation of the system call *set_sched_policy*, which changes the current scheduling policy used in the system. Currently this system call only support two scheduling policies: round robin and FCFS. Library *libuser.a* contains the implementation of the wrapper for this system call and library *libschedperf.a* contains the implementation of the system call. The interface is the following:

```
int set_sched_policy(int policy)
    policy: scheduling policy to use: 0 (round robin) or 1 (FCFS)
    returns: -1 if policy parameter is not valid and 0 if OK
```

5.1.2 What do you have to do

- Update function *init_sched* to call function *init_sched_policy*.
- Implement functions *get_task_stats* and *get_task_list* which are needed by the FCFS functions. These functions, given a process, have to return a pointer to the statistics field in its *task_struct* and a pointer to the *list_head* field. This way, FCFS code is independent of the particular name that you have used for those fields. The interface is the following:

```
struct stats * get_task_stats(struct task_struct *t)
    returns: a pointer to the statistics field in task t
```

```
struct list_head *get_task_list(struct task_struct *t)
    returns: a pointer to the list_head field in the task t
```

- Complete the implementation of the system call *set_sched_policy*. It is necessary just to modify the system call table to reference function *sys_set_sched_policy*. The identifier of this new system call is 17.
- Adapt your code to use the scheduling interface variables instead of calling directly the round robin functions.
- Check that your code runs properly when the active scheduling policy is round robin.

5.2 Support to consider blocked processes

We want to evaluate scheduling policies while dealing with blocked processes too. For this reason we have added a blocking system call and the code necessary to block and unblock processes. The new system call emulates a read from console operation. When a process executes this system call it blocks as many clock ticks as bytes has requested. Clock interrupt routine counts the amount of ticks that a blocked process remains in this state and unblocks it when it reaches the blocked time associated to the operation. We have defined a new processes list, *console_blocked*, which links all processes that are blocked due a console read operation. This means that all alive processes (except the *idle* process) can be in run state (the *current* process), or in the *readyqueue* or in the *console_blocked*.

5.2.1 What is already done

- Console device initialization, including declaration and initialization of *console_blocked* list. Function *zeos_console_init* implements this code and you only have to call it as part of the system initialization code.

```
void zeos_console_init()
```

- Implementation of system call *read*: wrapper code is in library *libuser.a* and *sys_read* function is in library *libschedperf.a*. Function *sys_read* has to be invoked through the system call table. It just checks that the parameters are right and invokes the function that reads from console. The interface of the read system call is the following:

```
int read(int fd, char *buf, int nbytes);
    fd: virtual device, currently only 0 is valid .
    buf: currently this parameter is ignored.
    nbytes: represents the number of blocked ticks.
    returns: -1 if error, nbytes otherwise
```

- Implementation of console read operation: implementation of reading from console is based on 3 functions, which are part of library *libschedperf.a*: *sys_read_console*, *zeos_read_console_emul* and *zeos_update_read_console_emul*.
 - *sys_read_console*: this function just calls function *zeos_read_console_emul*, which is the function that actually emulates console reading. The interface of this function is the following:

```
int sys_read_console(char*buf, int size);
    returns: size
```

- *zeos_read_console_emul*: this function blocks the process that is reading from console, by calling function *block_process*. The interface of this function is:

```
int zeos_read_console_emul(int nticks);
    returns: nticks
```

- *zeos_update_read_console_emul*: this function must be called from the clock interrupt routine. It unblocks a process if it is necessary. It emulates sequential I/O and serves request in a FIFO way. This means that, until the read of the first process ends (its blocking time ends), it does not start the operation of the following blocked process in the queue. The interface of this function is:

```
void zeos_update_read_console_emul();
```

5.2.2 What do you have to do

- You have to add to main routine in *system.c* the code to initialize the console (just a call to function *zeos_console_init*)
- You have to implement functions to block and to unblock processes. These functions are already used inside the console management code. This is the code that you should implement:

```
void block_process(struct list_head *block_queue) {
```

```

struct task_struct *t = current();
struct stats *st = get_task_stats(t);

    st->system_ticks = get_ticks()-st->elapsed_total_ticks;
    st->elapsed_total_ticks = get_ticks();
    update_process_state(t, block_queue);
    sched_next();
}

```

```

void unblock_process(struct task_struct *blocked){
struct stats *st = get_task_stats(blocked);
struct list_head *l = get_task_list(blocked);

    st->blocked_ticks += (get_ticks()-st->elapsed_total_ticks);
    st->elapsed_total_ticks = get_ticks();
    update_process_state(blocked, &readyqueue)
    if (needs_sched()) {
        update_process_state(current(), &readyqueue);
        sched_next();
    }
}

```

Notice that this code includes the lines necessary to update the stats of the process with its blocking time.

- You have to add to the clock interrupt service routine a call to function `zeos_update_read_console_emul`.
- You have to add to the system call table an entry for `read` system call. The identifier for this system call is 3.

5.3 Experiments

Next step is to evaluate the performance of several processes depending on the scheduling policy used. For each process we want to measure: time that the process spends executing user code (field `user_ticks` in the process stats), time that the process spends executing system code (field `system_ticks`), blocked time (field `blocked_ticks`), ready time (field `ready_ticks`) and total time (which is the sum of the rest of values).

5.3.1 What do you have to do

- You have to define several workloads (not less than 3), implement them in file `user.c`, and take the performance measures indicated using both scheduling policies (FCFS and round robin).
- You have to represent the results using some graphs that show execution time for each process (including idle process). File `soa_schedPerf` is a spreadsheet that you can use for this purpose.
- You have to submit:
 - Your version of ZeOS code including all the modifications described in this document and all user code that you have implemented to perform the experiments
 - File `soa_schedPerf` updated with your results and graphs
 - A plain file describing the workloads that you have implemented and containing a discussion about the results for each workload and each scheduling policy.

5.3.2 Some issues to consider when defining the workloads

For our scheduling performance analysis we only have to consider execution bursts: CPU bursts and blocked bursts. It is not necessary to implement complex tasks. Each workload has to be composed of several processes (not less than 3) and you can test several combinations: all processes being CPU bounded, all processes being I/O bounded, combination of both types of processes, large I/O bursts and short CPU bursts, etc...

6 PROCESS COMMUNICATION

The goal of this laboratory work is to complete the implementation of several server schemes (see subsection 6.2) and to evaluate the performance of each of them (see subsection 6.1).

Package `soa_sockets.tar.gz` contains the following files:

- `serverSocket.c`: implements a sequential server. Function *doService* implements the service and the communication with a client. The parameter of this program is the port number used by the server for the communication with clients. The command to launch this process is (assuming that the executable is in the current working directory):

```
./serverSocket port
```

- `clientSocket.c`: implements a client. Its parameters are number of messages to send to the server, server hostname and port number that the server uses to receive client messages. It connects with the server and executes a loop to implement the communication with the server. The number of iterations of this loop is the number of messages that the client has to send, according to its first parameter. During the execution, a client shows on standard output all the messages it receives. At the end of the execution, it shows on standard error output its total execution time. The command to execute this program is:

```
./clientSocket msgNumber serverHost serverPort
```

- `socketMng.c`: this file should implement the socket creation, socket delete and socket configuration code. *Currently it only implements clientConnection and deleteSocket functions. You must add the implementations for createServerSocket and acceptNewConnections functions.*
- `launchClient.c`: the purpose of this code is to manage the concurrent execution of several clients. The parameters are number of clients to lunch and parameters for each client (all clients will receive the same parameters). This code executes a loop that, for each iteration, creates a new process. Each child process, redirects its standard error output to a new file, and mutates to execute the `clientSocket` program. The main process controls the number of concurrent clients and keeps in file *launch_info* the maximum concurrency degree that it is reached, that is, the maximum number of clients that are alive simultaneously. In this file it also stores the total time required to execute all clients. The command to execute this program is:

```
./launchClient clientNumber msgNumber serverHost serverPort
```

- `Makefile`: to compile all the files

6.1 Performance evaluation

Our goal is to find out which of the server schemes is the most efficient, considering both the time spent to serve a client and its scalability properties, that is, how many concurrent clients it is able to serve without losing performance. To meet this goal, it is necessary to execute several tests changing the amount of concurrent clients, by configuring both parameters the amount of launched clients and the length of each client. Be aware that given a server scheme, concurrency degree depends on the value of these two parameters.

For each test, we are going to take the following performance metrics:

- *Average execution time for each client.*
- *Number of successfully served clients per time unit.*

For each server scheme, you should execute several tests increasing the concurrency degree in order to find out the saturation point for each server. That is, that concurrency degree at which the quality of service level starts deteriorating.

Take into account that, for each user, the system can limit the maximum number of processes and the maximum number of open files. Thus, maybe you need to modify these limits in order to execute enough processes to saturate the servers. Command `ulimit -a` shows the current value of these limits for your current session. In order to modify these values for all the sessions started in the machine you can modify the file `/etc/security/limits.conf` by adding the following lines:

```
@alumne hard nproc max_num_processes
@alumne hard nofiles max_open_files
```

Where `max_num_processes` stands for the desire limit for maximum number or processes and `max_open_files` stands for the desire limit for the maximum number of open files. Next session started in the machine will use these new limits. Notice that it is necessary to be root in order to modify this file.

6.2 Server schemes to implement and to evaluate

- 1) *Sequential server*: this scheme is already implemented in file `serverSocket.c`. You only have to complete its implementation adding to `socketMng.c` file those functions to create and initialize sockets.
- 2) *Unbounded concurrent server*: update the server code to enable it to serve concurrent clients. The changes required are to implement the `doServiceFork` function and to call this function each time the server accepts a client connection (instead of calling `doService` function). Function `doServiceFork` just creates a new process which executes the `doService` function and then ends the execution. In this scheme the number of concurrent accepted clients is unbounded: the server accepts all the client requests received.
- 3) *Bounded concurrent server*: this scheme is a variation of the unbounded concurrent server, which controls the number of concurrent accepted clients. The goal is to accept only the clients that it is able to serve without suffering performance deterioration. In this scheme, the server have to keep a counter of its current alive children and when this counter reaches an upper bound it has to stop accepting new client connections and has to block until a child ends the execution. You have to choose the value for this upper bound and define a constant in the server code or add a new parameter to the main program.

- 4) *Extra bonus: concurrent server using threads*: this scheme implements a concurrent server but using pthreads instead of using heavyweight processes (created with *fork* system calls). You can work on the unbounded concurrent server and update it to use pthreads. Compare the saturation point reached with this version with the saturation point reached using traditional processes.

6.3 Submission

You must submit all your source code, a spreadsheet showing your experiments results and a plain file with the description of the tests and a discussion of the results. The spreadsheet has to show for each server scheme a graph representing the results depending on the concurrency degree. For each graph you have to clearly describe the parameters that characterize the experiment: server scheme, number of launched clients, number of messages per client, maximum concurrency degree....

7 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
```