Multitasking ZeOS

Yolanda Becerra, Juan José Costa and Alex Pajuelo

Curs 2013–2014 - Quadrimestre tardor

CONTENTS

1 PREFACE

This documents describes the steps to follow to make ZeOS become a multiprocessing operating system. The current implementation is monouser and monoprocess.

Before carrying out all the work described in this document, it is very important that the student fully understands how the current implementation of ZeOS works. For this, it is adviseable that the ZeOS document must be read carefully following the text with the analysis of the OS code.

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.
- Remember some of the concepts needed.

2.1 Getting started

This section describes the step to set up Zeos. The following steps are performed supposing that the student works with an standard installation of Ubuntu 10.X (skip to the 2nd step if you are in the laboratory):

1) Prepare the development environment:
   a) Install basic development packages

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

   b) Install package for assembler x86: as86

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

   c) Download and install Bochs 2.3.
2) Download and install ZeOS:
   a) ZeOS is available as a compressed file (.tar.gz) at the web page.
   b) To install it, execute:

```
> tar zxfv zeos.tar.gz
```

3) Test the environment:
   a) Generate your ZeOS (zeos.bin):

```
> make
```

   b) Execute it using bochs:

```
> make emuldbg
```

   c) If everything worked well, a window like the one in Figure 1 will appear. Meaning that your emulated computer is about to boot your ZeOS image. Press c and INTRO to check that it works.

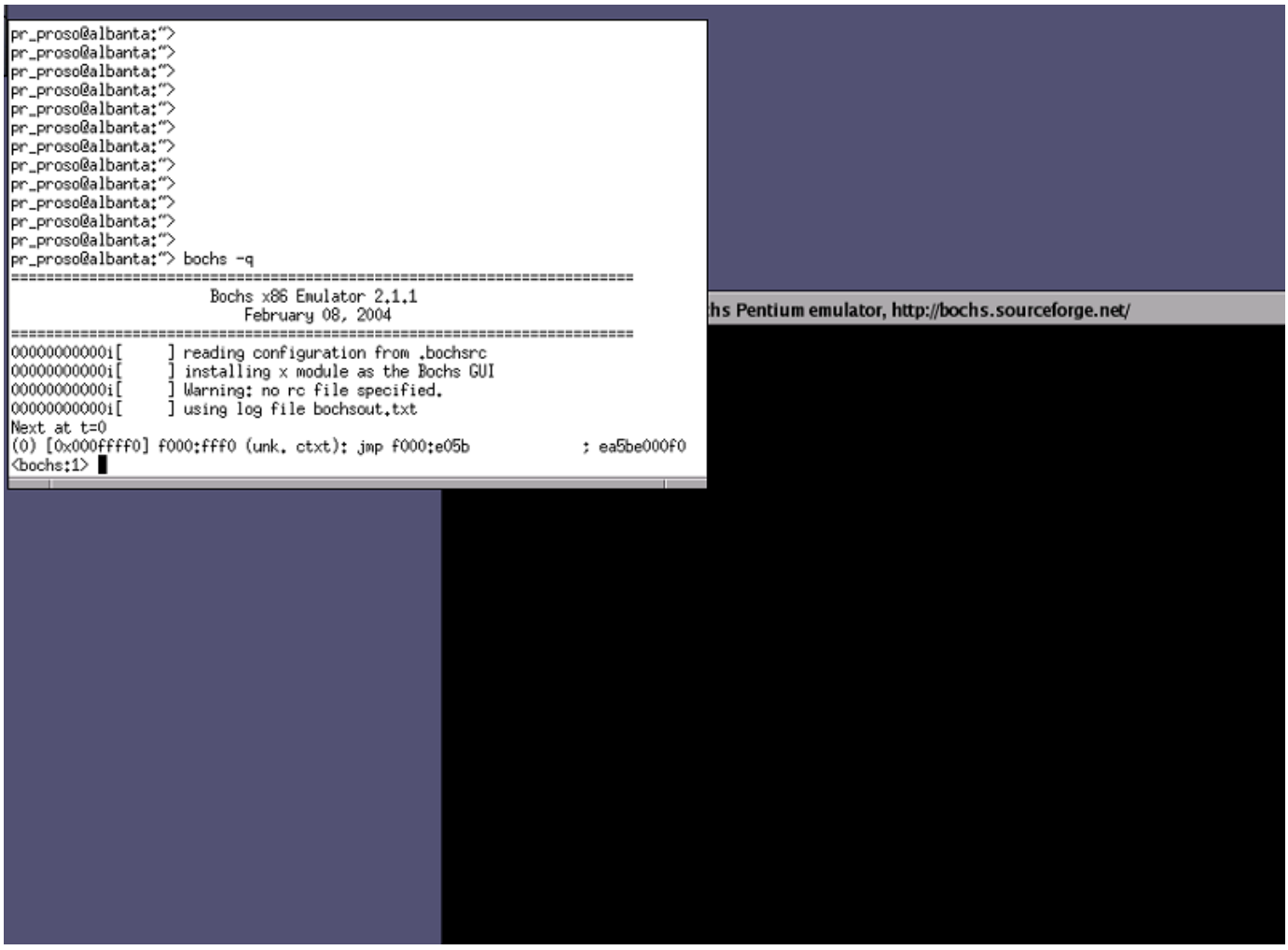

Fig. 1. Working environment with the Bochs commands window (left) and Bochs emulation window (right)

2.2 Working environment

When an OS is being developed, a working environment is needed and it is usually offered by another operating system. In this case, to develop ZeOS we will use an Ubuntu system with the following configuration:

- OS. Ubuntu 10.04
- Compiler. GCC 4.0.3
- Emulator. Bochs version 2.3 (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. Not all shells support this, so, if it is not available, the commands would be:

- Compress: `tar cvf file.tar files_to_unify` and then `gzip file.tar`
- Extract: `gunzip file.tar.gz` and then `tar xvf file.tar`

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

2.3 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. More specifically, it is assumed that you are able to:

- Write fairly complex programs in C language.
- Write fairly complex programs in Linux i386 assembler language.
- Add assembler code to a C file.
- Modify a Makefile to add new rules.

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.

2.4 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 2.3**.

2.4.1 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.

2.4.2 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: using an external debugger (such as gdb) or using an internal debugger that is part of the emulator. The two options are *exclusive*, namely, the Bochs executable can support only one. Both versions are available in the laboratory (bochs and bochs_nogdb), so you can choose whichever version you prefer.

Compiling Bochs to support debugging

This section is only needed if you plan to compile it yourself at home. The two binaries provided in the laboratory (bochs and bochs_nogdb) have been compiled using the information presented here.

In order to use the Bochs debugging facilities it is necessary to activate some debugger options during the compilation of the Bochs source code, 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
$ make all install
```

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

```
$ ./configure --enable-debug -enable-disasm
$ make all install
```

2.4.3 Bochs configuration file

Bochs provides a configuration file called *.bochsrc*. This file defines the features of the emulated computer. We will focus on the following two points:

- ***Image location***. Line 134 defines the file zeos.bin as ZeOS image to boot. The file name 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
```

- ***External debugger configuration***. A line has been added at the end of the file to configure the use of the GDB debugger. If Bochs has been compiled to use the external debugger without this additional line, it will execute without any debugging support.

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

It should be highlighted that this line is only interpreted when the Bochs version is compiled to use the external debugger. **By default, this line is deactivated**, so the file *.bochsrc* is valid for the two Bochs versions installed in the laboratory. **If you want to use the gdb, you should activate this line.**

Once you have modified your *.bochsrc* to your needs, you can execute Bochs to load your own image. You can execute the bochs version using the command[1]:

```
$bochs -q
```

This will start a new window (emulation window) with the typical screen output (some bios information and some messages that the operating system writes) that you would see after starting a physical computer.

1. The target *emul* from the Makefile has the same effect.

2.4.4 Controlling execution through internal Bochs debugger

- Compile the ZeOS code with debug symbols, using "-g" flag in CC (it is enabled by default in the Makefile).
- Ensure that the configuration line gdbstub has been *disabled* in the *.bochsrc* file.
- Execute the bochs version without GDB[2].

```
$ bochs_nogdb -q
```

Once the version of bochs with the debugger option is executed, you should see two windows like the ones shown in previous Figure 1: the emulation window and the commands window (the same where bochs has been executed).

The commands window displays a prompt in which commands can be introduced. This prompt shows information about the memory address and its contents that the debugger is about to execute. A summary of the commands that can be executed may be found at http://bochs.sourceforge.net/doc/docbook/user/internal-debugger.html.

2.4.5 Controlling execution through an external GDB debugger

To use GDB (GNU debugger) to debug your operating system, follow the steps below:

- Compile the ZeOS code with debug symbols, using "-g" flag in CC (it is enabled by default in the Makefile).
- Ensure that the configuration line gdbstub has been *enabled* in the *.bochsrc* file.
- Execute the bochs version with the external debugger activated[3].

```
$bochs -q
```

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

Figure 2 shows the emulation window (left) and the window (right) where bochs has been started.

- After executing bochs, a new terminal is needed to execute GDB or a front-end to GDB (for example ddd). GDB accepts an executable program as the argument to be debugged. The system binary file of our Zeos will be usually used.

```
$gdb system
```

- Connect GDB to Bochs by executing the command below in GDB (the port should be the same as the one you use in *.bochsrc*).

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

- Add user symbols.

2. The target *emuldbg* from the Makefile has the same effect.
3. The target *emul* from the Makefile has the same effect.

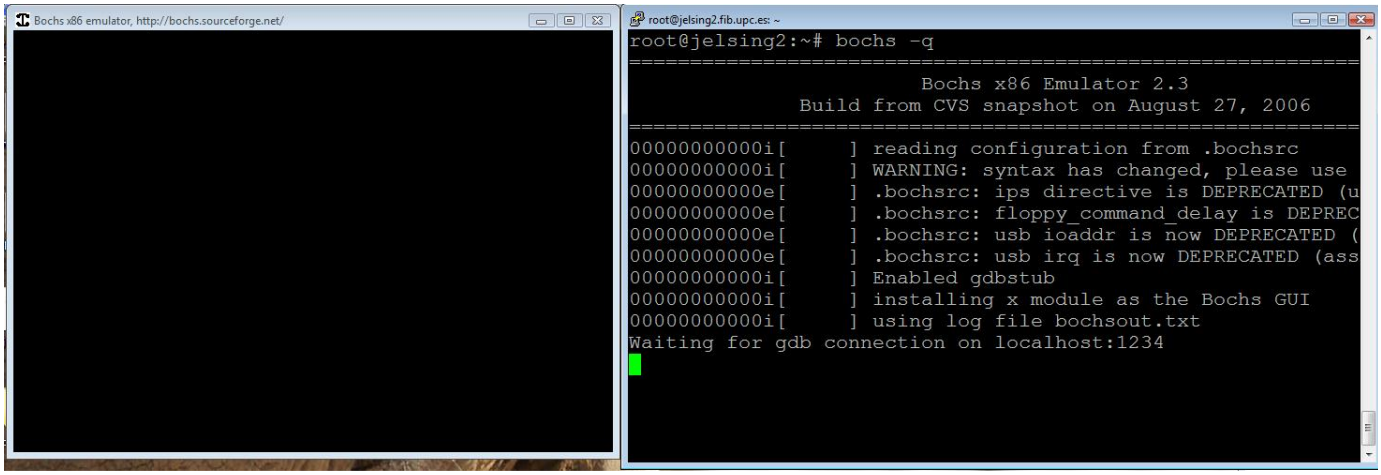

Fig. 2. Emulation window (left) and commands window (right) waiting for a GDB connection appearing when bochs starts with the external debugger enabled.

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

- You can now add breakpoints, continue the execution, etc. You can find a reference guide with the main GDB commands on the following link: http://refcards.com/docs/peschr/gdb/gdb-refcard-a4.pdf

2.5 Frequently used commands

Table 1 presents some of the most frequently used commands for bochs and gdb debuggers.

3 First steps

The user.c file is the place in which the user code must be written. **We strongly recommend that you only modify the user.c file.**

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

- Use the debugger to control and view information about your code.
- Visualize the generated object code.
- Locate variables and functions in object code.
- Modify the user code.
- Mix assembler and C in your code.

Bochs	GDB	Description
help		Show the commands that can be executed.
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	Execute *num* assembler instructions. Execute a single one if no *num* is provided.
next [*num*]	nexti	Execute *num* assembler instructions. Execute a single one if no *num* is provided. If the instruction is a *call*, the whole function is executed.
break *address*	**b** **address*	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", it will insert a breakpoint in the first code line of the user process.
r	**info r**	Show the content of the registers in the mode currently being used (user or system)
print-stack [*num_words*]		Print *num_words* from the bottom of the stack. If *num_words* is not specified, the default value is 16. It is only reliable if you are in system mode, when the base address of the stack segment is 0.
info tab		Show paging address translation.
quit		Exit debugger.

TABLE 1
Frequently used commands for bochs and gdb

3.1 Where is ZeOS hanged?

In this section you must prove that after running the ZeOS image it is executing the infinite loop in the user code. To do that you must:

1) Generate a ZeOS image. Check whether there are any warnings when it is compiled. Correct any errors that may have occurred.
2) Run the bochs debugger and execute this image.
3) When the ZeOS seems hanged, which instruction is executing?
4) Where is that instruction in the code?
   - To interpret the code execution, it is useful to know the memory location of each section of the image. For that, the commands *objdump* and *nm* can be used.
   - The objdump command shows the compiler-generated code and the corresponding memory addresses, which can be useful to insert breakpoints (Figure 3 shows the result of using objdump with the system file). The nm command shows the location of variables and functions. For instance, Figure 4 shows that the task struct array is located at address 0x11000.

3.2 User code modification

- Modify the main() of the user.c file as Figure 5, adding some local variables and a call to an *outer* function.
- Add the functions *inner* and *outer* to the user.c file just before the *main* as in Figure 6.
- Generate a new ZeOS image. Check whether there are any warnings when it is compiled. Correct any errors that may have occurred.
- Execute the command "objdump -d user.o" to see the code in the assembler language of the file user.c. Note that the addresses of the symbols that appear are in PIC (Position Independent Code) and that they start from zero.

```
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    $0xfffffff0,%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. 3. "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. 4. "nm system" output sample

- Now execute "objdump -d user". *Do the addresses match? Why?*
- Run the whole program (*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 step by step. To do this, add a BREAKPOINT to the first instruction of the inner routine and continue the execution using the continue command. Check that the execution stops at the breakpoint line. Then, continue the code execution line by line (step and/or next[4]).

Note: Adding a BREAKPOINT is often the only useful way to reach to a subroutine address. In this case, the number of STEP commands that may be needed to reach the inner() routine is extremely high. In the ZeOS project, one BREAKPOINT is the only way to ensure that the execution jumps to an interrupt service routine (ISR).

- Examine the content of the registers.
- Which debugger command enables you to see the content of the memory? Does it work to see the content of the acum variable from the user.c file?

4. It has been seen that after typing Ctrl+C, Bochs is unable to continue the execution with the debugger instructions step or next. The commands continue, stepi or nexti must be used

```
int __attribute__ ((__section__(".text main")))

main()
{
        long count, acum,
        count = 75;
        acum = 0;
        acum = outer(count);
        while (1);
        return 0;
}
```

Fig. 5. Modification to the main() function in the user.c file

```
long inner(long n)
{
        int i;
        long suma;
        suma = 0;
        for (i=0; i<n; i++) suma = suma + i;
        return suma;
}
long outer(long n)
{
        int i;
        long acum;
        acum = 0;
        for (i=0; i<n; i++) acum = acum + inner(i);
        return acum;
}
int __attribute__ ((__section__(".text.main")))
main()
. . .
```

Fig. 6. New functions *inner* and *outer* to the added to the user.c file

- How can you see the value of acum in an easy way? (TIP: think about where C stores the returning value of a function)
- How can you see the parameters passed to the outer function?

3.3 Use of inline assembler

Add one function to the user.c file using the following header:

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

This function must perform (in assembler) the same as:

```
return par1+par2;
```

See the document in the bibliography about how to add assembler code to a C program. To do the sum, you must access the parameters (remember that they are on the stack), add them and return the result. Two different versions will be written:

- The entire code of the function will be written in assembler language. This means that you must review the way a value is returned in assembler; (in which register will the result have to be placed?). You cannot use the input/output parameters of inline assembler.
- Use assembler inline. A local variable must be defined in C and the result of the addition will be put in this variable. See the appendix to ascertain how to modify a defined variable from C in an assembler code. In this case, you must use the input/output parameters of inline assembler.

Add a call to the function add() in the main() function of the user.c file by putting the result of the addition in a local variable.

- Check with the debugger that both versions work correctly.
- Which address did you enter in the Bochs debugger to access the variable in which you put the result of the add function?

3.4 Additional questions

Try to understand the relevant parts of the boot mechanism (bootsect.S) and system initialization (system.c). You can draw a structure diagram of the files and their contents. You should be ready to answer the following questions:

1) What are the main differences between real mode and protected mode?
2) In which mode is the system when it is booting?
3) What are the execution privilege levels?
4) What is the bootloader? In which part of the disk is it found?
5) What is the BIOS? What is it for?
6) What steps does ZeOS Makefile perform? What are the uses of the various compilation and linkage options?
7) Optionally, you can try to scroll the screen if the cursor surpases the maximum coordinates.

4 MECHANISMS FOR ENTERING THE SYSTEM

At this point you must have read the ZEOS document corresponding to the introduction and the mechanisms to enter the system.

In this practical session, you will complete the ZeOS code to match the ZEOS document and implement the clock interrupt management and a new system call: gettime.

4.1 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 *system_call_handler* routine.
- Initialize the IDT with the handler
- 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 *errno* and *perror* function.

4.2 Clock management

The clock management will display the seconds that have elapsed since the boot process. Figure 7 shows the expected result with the time displayed in a fixed place on the screen.

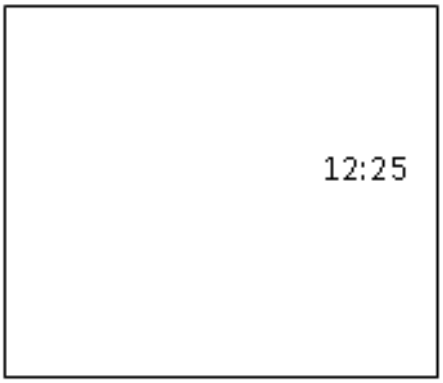

Fig. 7. Clock interrupt

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

- Initialize the entry of the clock interrupt in the IDT. The clock interrupt is contained in entry 32.
- Write the handler.
- Write the service routine.
- 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.

4.2.1 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.

4.2.2 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();
```

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, whether you return to the user mode, etc.

4.3 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.

5 PROCESS MANAGEMENT

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.

5.1 Complete ZeOS code

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

- Adpat 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.

5.1.1 Task_struct modifications

You should add to the task_struct definition all the fields required to implement the process management. Even it is not needed for the current features requested by ZeOS, we ask you to add a new field named `quantum`, that will hold the quantum of the process. This value must be used by the process scheduling policy to control the maximum consecutive time that the process can use the CPU, if there are other ready processes.

Notice that this field **must not** be modified in your code during the life of the process but it could be modified externally.

5.1.2 Initial process initialization

As part of the initialization code it is necessary to initialize both the idle process and the init process. 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.

5.1.3 Implementation of the context switch between processes

You will implement the task_switch routine and you will use the idle and init processes to check that it works (for example, executing the init process at the beginning and after some ticks make a task_switch to the idle).

5.1.4 Implementation of the fork system call

The stack of the new process must be forged so it can be restored at some point in the future by a task_switch. In fact the new process has to restore its hardware context and continue the execution of the user process, so you can create a routine *ret_from_fork* which does exactly this. And use it as the restore point like in the idle process initialization.