Zeos

Yolanda Becerra, Juan José Costa and Alex Pajuelo

Curs 2013–2014 - Quadrimestre tardor

PREFACE

The aim of this document is to serve as a basic documentation of an operating system called ZeOS, 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.

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. Both high-level (C) and low-level (assembler) programming languages will be used in order to add more functionality 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 borne in mind:

- 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 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)

2 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 in fact be useful in explaining how to build a Linux OS.

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. The *Makefile* will guide you through the process, compiling all the source files and linking them together to build a final bootable image (a file called *zeos.bin*). This process is explained in detail in the following subsections.

2.1 Contents

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:

1) Source files written in C language, whose extension is .c
2) Source files written in Intel 80386 assembler language with preprocessor sentences, with an .S (capital S) extension. The .c and .S files are the only ones that add code to the OS.
3) Scripts used by the ld linker to combine the various files in a single binary file, with an .lds extension.
4) Header files (extension .h) located in a dedicated directory, as occurs in Linux.
5) The Makefile that will guide you through the steps to build the OS. It is the only file without an extension.

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.

After the make process you should obtain a single file called zeos.bin with the bootable image of your ZeOS (which could be copied to a floppy disk to test that it effectively boots up in your computer).

All files that add code to ZeOS (.c and .S files) can be divided into several groups, depending on the final position of each file in the image of the OS. Figure 1 shows the final snapshot that the image of the OS should have once it is built[1]. It shows three main blocks:

- The ***boot sector* block**. This block only comprises one file: bootsect.S. It must weigh exactly 512 bytes, since it has to be fitted in sector 0 of a conventional floppy disk.

1. Some of the file names that appear in Figure 1 may not match the ones from your currently downloaded source code

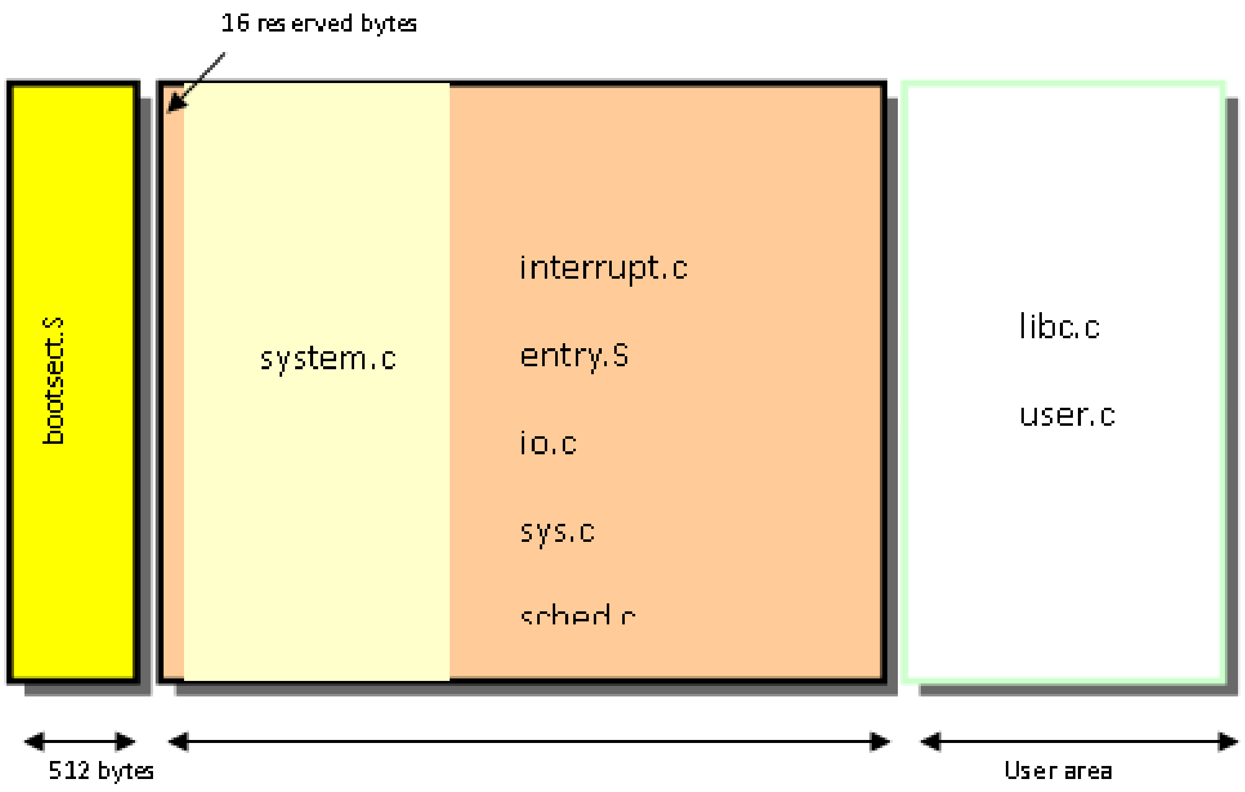


Fig. 1. Snapshot of Zeos

- The *system area* **block**. This block is the largest and contains the main files of the OS. This block is placed in a part of the memory that guarantees it will be executed with the processor's maximum privilege level.
- The *user area* **block**. This block has a user program that can make use of the operating system services (use system calls). The code inside is executed by the processor with a minimum privilege level.

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.

Figure 2 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, to store the user code size and other necessary data for managing the user code; then the system code (.text); the read-only data section (.rodata); the initialized data (.data); the not 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) wich must be highlighted because it will contain the task_struct array.

```
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. 2. system.lds file. Linker script file for the system image.

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

Fig. 3. Task_struct array annotated with the section .data.task (sched.c)

2.2 Booting

The three blocks described in the previous section are appended to each other to create a single binary file with the content of all ZeOS operating system. If this file is copied in binary format to an unformatted floppy disk, beginning with sector 0, an image will be obtained of the OS that will boot automatically. It will be possible to run it on any computer with a floppy unit that can boot from a floppy disk.

The booting operation is simple. First turn on your computer. Before the OS is executed, a program that is placed in a read only memory (ROM) area will be executed, called the BIOS. This program checks that the PC is working properly. It is possible to see how the memory, the disk and other devices are checked during this process. It then looks for the preconfigured booting device and tries to access it. It does so by loading sector 0 from this device 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. Since ZeOS is very simple, the loader is designed to fit into 512 bytes and the BIOS will fully load it into the memory.

Once this 512 bytes of the boot sector are copied to the memory by the BIOS, it transfers the execution to the first byte of this small block. The boot loader is executed at this point. The main task of this boot sector code is to finish loading what is left in the image of the OS that is still on the floppy disk and load it into the memory. Basically the system and user areas. Finally, it transfers the execution to the first byte of the system code and starts the execution of your ZeOS.

2.3 Image construction

The way the OS image is built will be explained backwards. It will be assumed that the three blocks (boot sector, system area and user area) have been obtained and it will be seen how to put them together into a single file like the snapshot in Figure 1. This process is done automatically using the Makefile that was 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 *syssize*. 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. That is why 16 bytes are reserved at the beginning of the system block, as can be seen in Figure 1. They are initially empty but the *build* program writes them with the sizes of the system and user images.

3 MECHANISMS TO ENTER THE SYSTEM

To execute code from the OS you must use the interrupt mechanism. Interrupts are events that break the sequential execution of a program and they can be classified as synchronous or asynchronous depending on when they are generated. 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.

Each interrupt is identified by a number between 0 and 255 and associated with a specific function using a system table known as an interrupt descriptor table (IDT). Each entry in the table has the information necessary to do the actions required when an interrupt is produced. Among other data, it contains the address of the routine to be executed (the **handler**) and the minimum code privilege level to execute it.

The interrupt id numbers are as follows:

0-31 are exceptions and unmasked interrupts. 32-47 are masked interrupts. 48-255 are software interrupts. For example: Linux uses id 0x80 (128) for the system calls.

These are the same numbers that appear in the Intel manuals. For more information, see chapter 4 of ***Understanding the Linux Kernel***.

To write an exception or interrupt (except the system call), you must:

1) **Initialize** the IDT entry that matches the exception or interrupt id number.
2) **Write the handler** to be executed when the interrupt or the exception is generated. The handler is the assembler code implemented to manage the call to the service routine.
3) **Write the service routine** to the interrupt or the exception, which will be the code to perform the associated service to the interrupt or exception.

An **additional** step is added for the system calls to isolate user codes from low-level and non-portable code. A function responsible for passing the system call parameters, generating the interrupt and recovering its result should be written for each system call. These functions will be named **wrappers**. The user code invokes any of this wrappers (as any other user function) that then causes the actual enter to the system.

In this section you will find:

- Hardware management of an interrupt.
- The code for managing exceptions.
- The code for managing the keyboard interrupt.
- The generic code for making system calls.
- The code for implementing a write system call.

The following sections include a how-to guide to add an exception, an interrupt and a system call. **Remember that they are almost the same, since they are all managed by the IDT.** Although they are described separately in this guide, it is important to understand that they are quite similar.

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 Hardware management of an interrupt

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

- The i index of the interrupt vector is determined and the corresponding IDT entry is accessed.
- The hardware verifies 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, a general protection exception is generated.
- The privilege level of the handler routine is checked to see if it is different from the current execution level (our case), in which case the stack will have to be changed.
- To make the change from user stack to system stack, the system takes the address of system stack from the TSS. Once the ss and the sp of the system are known, the hardware saves: 1) the values of the ss and esp of the previous stack in the new stack (the user stack in this case); 2) the value of the eflags (state word); and 3) the values of cs and eip (address that generates the interrupt). The resulting stack is shown in Figure 4
- The address saved in the i^{th} entry of the IDT is executed.

The hardware performs the aforementioned steps automatically.

Once completed the interrupt management code, the control must be returned to the instruction that generated the interrupt by executing the **iret** instruction. The iret instruction takes the value 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 in used before the interrupt happened (user stack in our case).

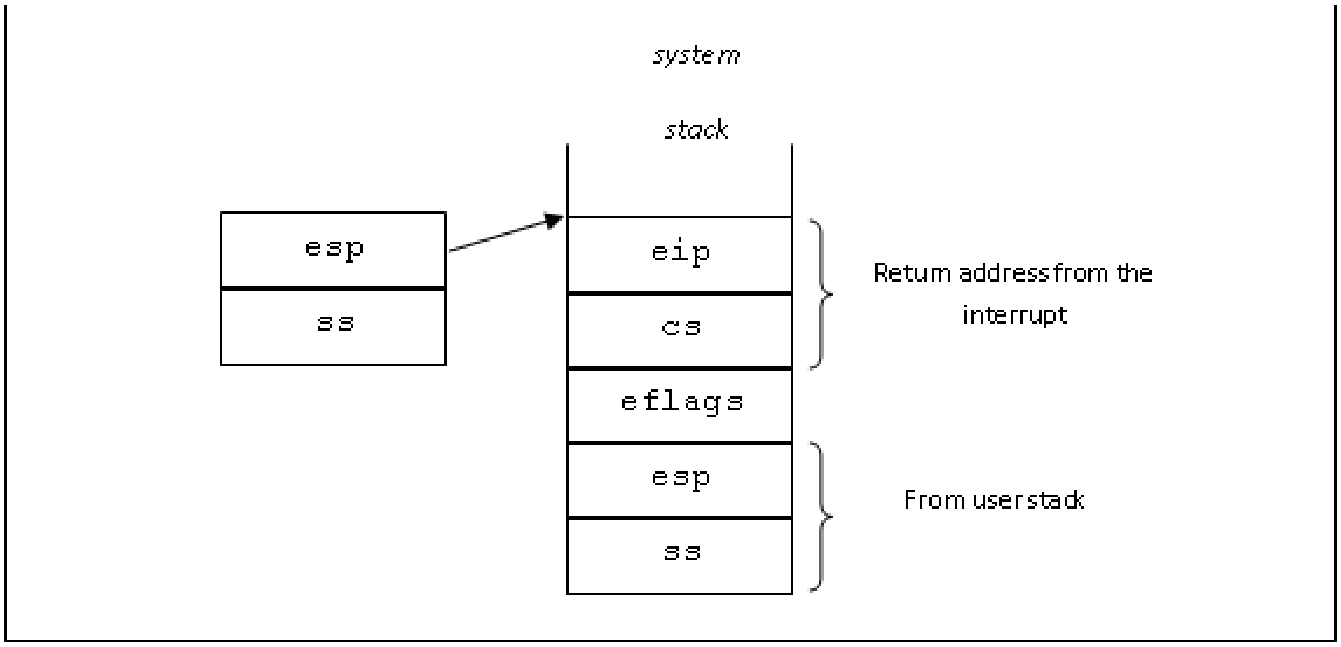


Fig. 4. State of the system stack when the execution of the interrupt handler begins

3.3 Function name conventions

The functions will be given similar names to make the code easier to follow. The handlers 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 "divide_error_routine".

As regards the system calls, the handler will be named "system_call" and the service routines "sys_namesyscall", for example "sys_write".

3.4 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
- entry.S: Entry system points (handlers)
- interrupt.c: Interruption service routines and exceptions
- sys.c: System calls

- device.c: Device dependent part of the system calls
- libc.c: System call wrappers
- io.c: Basic management of input-output

3.5 Programming exceptions

The system exceptions will be programmed in this section (except for the 15th exception, which is reserved for Intel). You will have to enter the system in order to catch an exception and execute the routine that you have created to handle it, as they are protected operations. The OS usually tries to recover from the exception and restore the system but in ZeOS the exception service routines are very simpl: the screen will show that an exception has been generated and specify which one, and the system will remain in an infinite loop state (it will never return to the user mode).

3.5.1 Exception parameters

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. Only the exceptions indicated in Table 1 receive these parameters. Figure 5 shows the system stack contents and the location pointed by the esp and ss registers when an exception with an error parameter is raised.

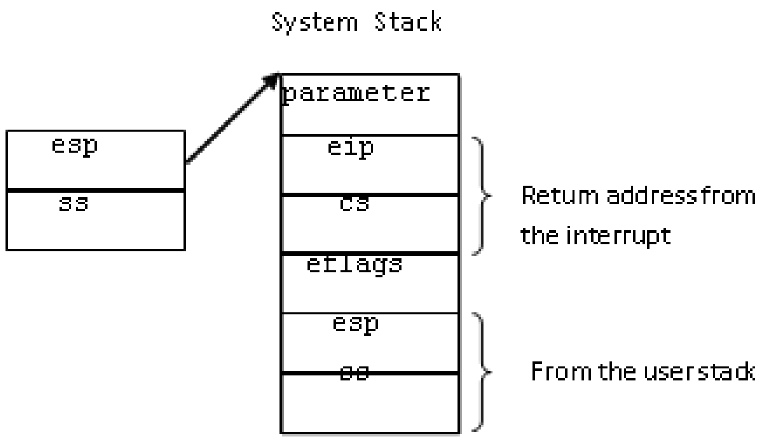


Fig. 5. System stack contents when an exception with an error parameter is raised

3.5.2 Initializing the IDT

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

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

The parameters required to write the exception are:

- `int position`. Entry in the IDT.

- `void (*handler)()`. Address of the handler that will handle the exception[2].
- `int privLevel`. Privilege level necessary to execute the handler. There are two possibilities: 0- Kernel, 3- User.

Table 1 shows the exception positions in the IDT, their names and the number of bytes of their parameters (or error code[3]) if they exist.

# 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 1
List of system exceptions

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

3.5.3 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** macro[4] can be used. The header is defined by calling it at the beginning of the code of the function and 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) Clear the error code (if the exception has one), removing it from the stack. Check Table 1.

2. If you do not know how to declare and use the function pointers, consult the C Programming appendix
3. 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
4. The corresponding annex shows how the macro mechanism of the assembler works

6) Return from the exception. Since it is not a "normal" return because it has to change mode, an **iret** rather than a **ret**[5] will be used for the return.

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

3.5.4 *Writing the service routines*

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

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

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

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

3.6 Programming interrupts

Figure 6 shows the main steps for the clock interrupt. This interrupt can 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 the first part to be executed. Inside this function we find a call to the clock_routine function who will make the real interrupt management. The steps for the keyboard interrupt are similar and will be programmed in this section.

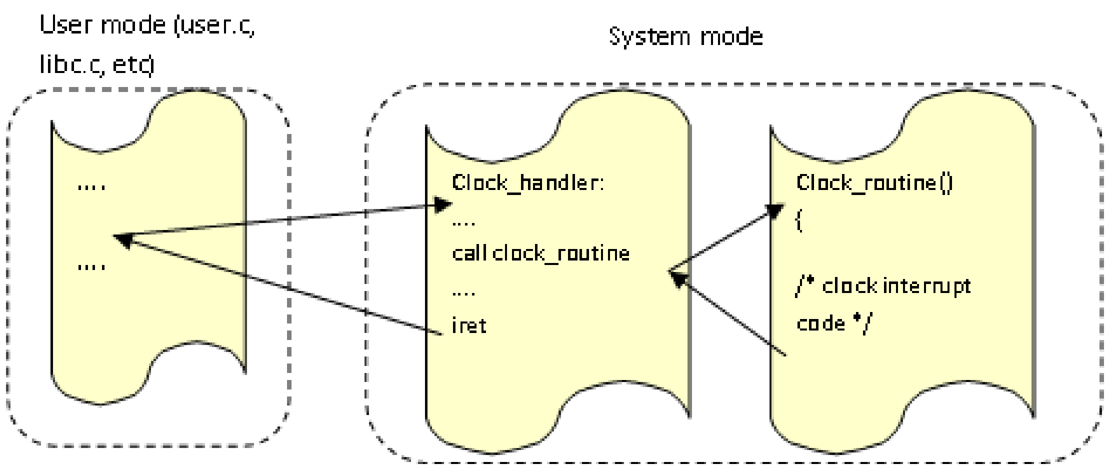


Fig. 6. Snapshot of the clock interrupt

3.6.1 *The keyboard interrupt*

The keyboard interrupt will display the character that corresponds to the pressed key. Figure 7 shows the expected result with the character displayed in a fixed place on the screen.

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

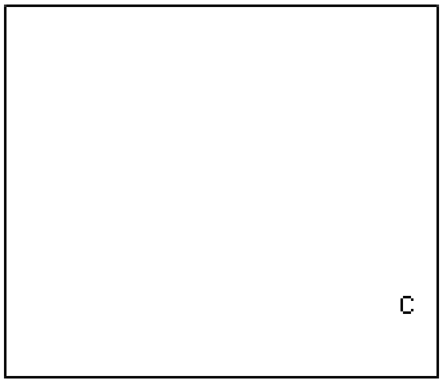


Fig. 7. Keyboard interrupt

To write an interrupt such as the keyboard interrupt, you must:

- Initialize the IDT as it is done for the exceptions. The keyboard interrupt is contained in entry 33.

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

- Write the handler.
- Write the service routine.
- Enable the interrupt. The key interrupt is a *masked* interrupt that is disabled in the code provided, which means that it is ignored by default. It is necessary to 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. For this reason, the best place to enable the interruptions is just after all the OS services have been started and before the return to user mode is performed.

3.6.2 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 are treating the interrupt. 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).

3.6.3 Writing the service routine

The 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 of the 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 8. Bit number 7 specifies whether it is a make or a break. Bits 0..6 contain the code to be traslated 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 printc_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.

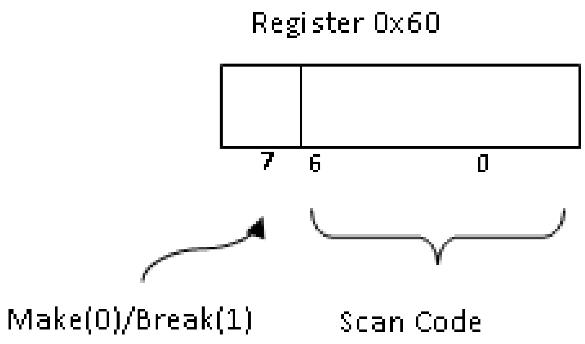


Fig. 8. Contents of the keyboard 0x60 register

3.7 Programming system calls

The first system call **write** will be written in this section. It will enable strings to be printed on the screen from user mode. 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: the 0x80 interrupt. This means that with one sole entry point (1 handler) there will be N system calls. In this section we will present all the common functions to all the system calls (and also the data structures) and the specific functions of the write call.

Figure 9 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, which you will call a **wrapper**, that is responsible for **passing parameters** between the user and the system, generating the **trap** (int $0x80) and processing the result.

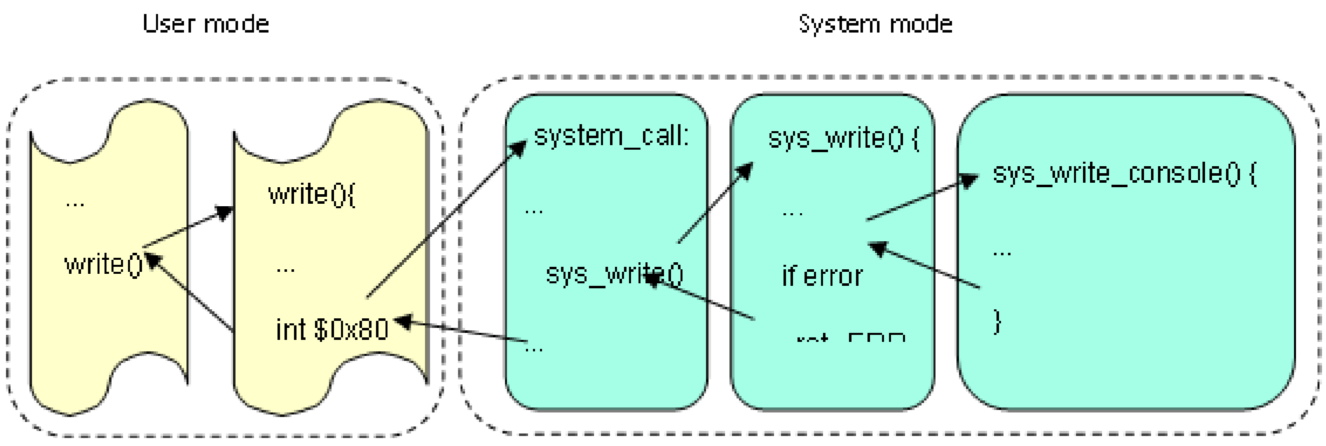


Fig. 9. Snapshot of a system call

Once the interrupt is generated, it is executed as another interrupt (the execution of the handler and the execution of the service routine). In this case, interrupt 0x80 does not need to be enabled since it is unmasked. However, the IDT does need to be initialized. As explained above, all system calls have a common handler, which will be named **system_call**. In this handler, specify the system call you want to make and execute the corresponding service routine (**sys_write** in the example).

3.7.1 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 have more devices added to it. The design will be similar to a real OS.

For example, the **sys_write** call (independent part) will call **sys_write_console** (dependent part), in order to print on the screen.

3.7.2 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 a specific system call. Thus, the **void perror()** function 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.

3.7.3 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.
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 3.7.2).
5) Return.

3.7.4 IDT initialization

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 priviledge level, the value for the third parameter will be 3:

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

3.7.5 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 register[6].
- 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 called **sys_call_table** will be defined in assembler. This table will be filled with entries like:

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

Bear in mind that non-implemented calls in a valid range must implement a call to **sys_ni_syscall**. This call will only return a negative identifier of the generated error. Each line is an entry of the table.

The instruction to execute the system call 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). Entry 4 points to the service routine (that will be written in section 3.7.6):

```
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
MAX_SYSCALL = . - sys_call_table
```

- Visit http://lxr.linux.no to look at the Linux declaration of the **sys_call_table** table.

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

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

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?
```

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

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

3.7.6 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 3.7.7).
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**.

3.7.7 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.

4 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 into 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 a, b and c, the compiler reserves memory for the maximum(a, b, c), instead of the sum of a, b and c 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 10, where both structures have been depicted sharing the same memory area, a region of 4096bytes.

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 array[7]. The declaration of the task array with the task_union is also provided:

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

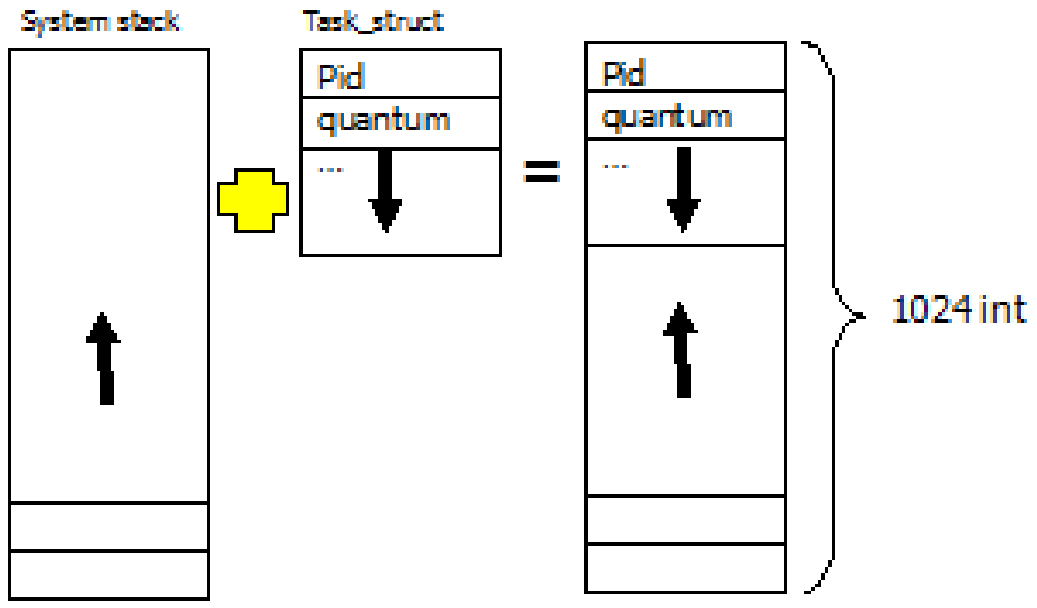


Fig. 10. The system stack and task_struct use the same page memory(4k)

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

The array is tagged with an attibute section to locate this array into the memory section named *data.task*. Figure 11 shows the placement of this task array in memory.

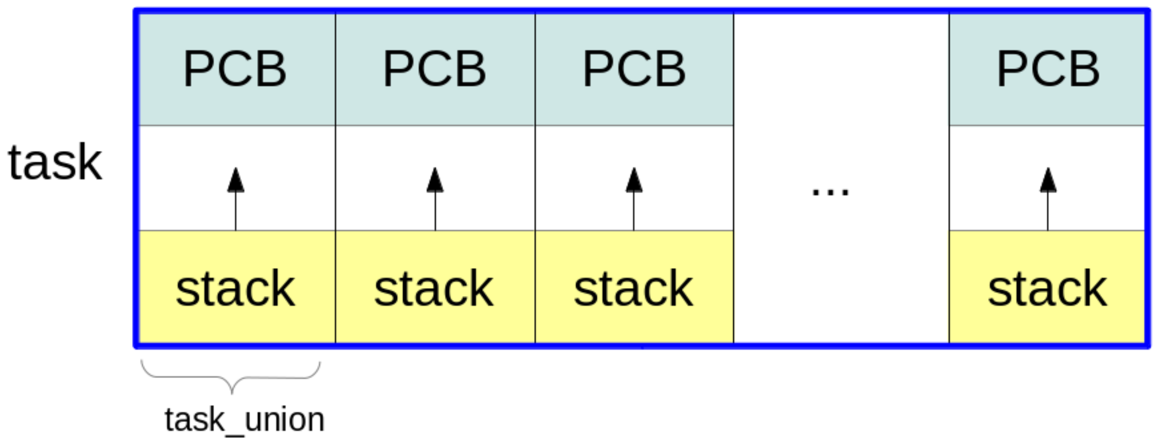


Fig. 11. Memory placement of the task array

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:

```
        void 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 address[8]. 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.

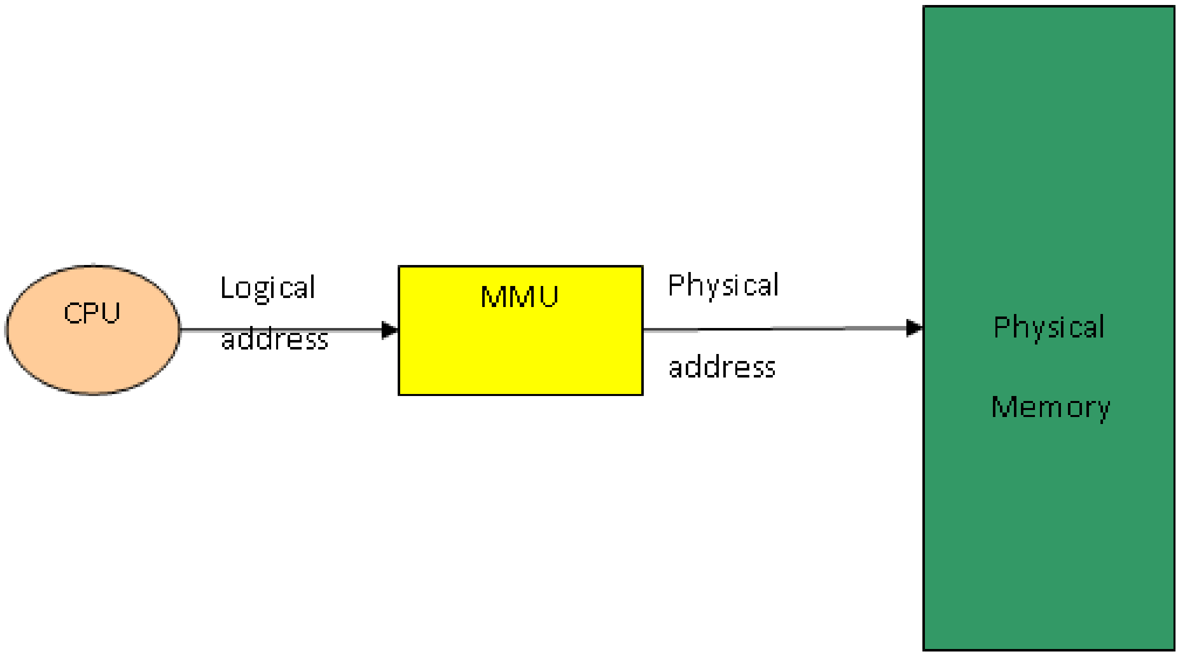


Fig. 12. Translation from logical addresses to physical addresses

8. 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

Figure 12 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 from logical to physical pages of the processes. **cr3** is a special register that keeps the base address of the page directory used by the MMU to do the current translations.

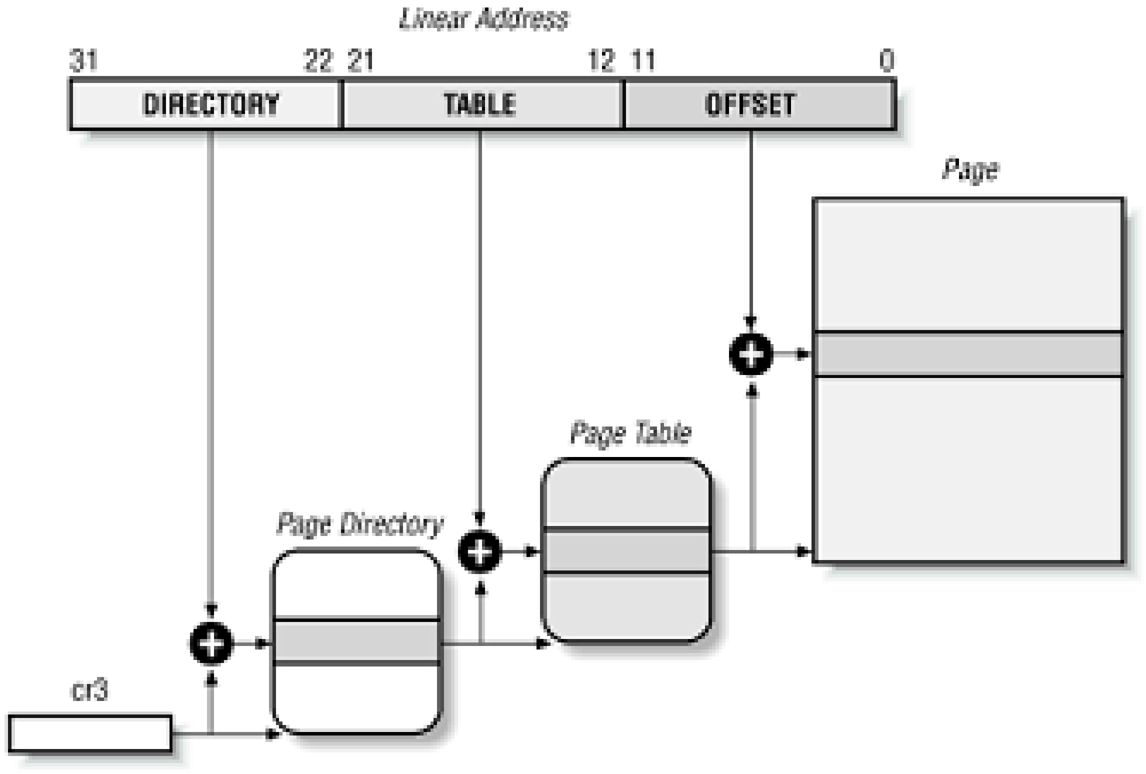


Fig. 13. Paging in the 80x86 architecture

Figure 13 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: page_number = (address >> 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).

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, and setting the field *dir_pages_baseAddr* of each *task_struct* to point to its corresponding page directory.

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 table[9], 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 Lookahead 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 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 Task State Segment (TSS)

The 80x86 architecture defines a specific segment called the task state segment (TSS) 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). **The TSS is mainly used to know the address of the kernel stack when making a user 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. You must understand how the TSS works because it is used in the context switch. **Read the steps on section 3.2** again that explain the hardware support for changing the execution mode (in an interrupt) to understand the use of the TSS.

Figure 14 shows how to use the ss0 and esp0 fields of the TSS. These fields always point to the bottom of the system stack. Hardware needs this information when switching to system mode.

4.3.5 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.

9. instead of an entry of a page directory

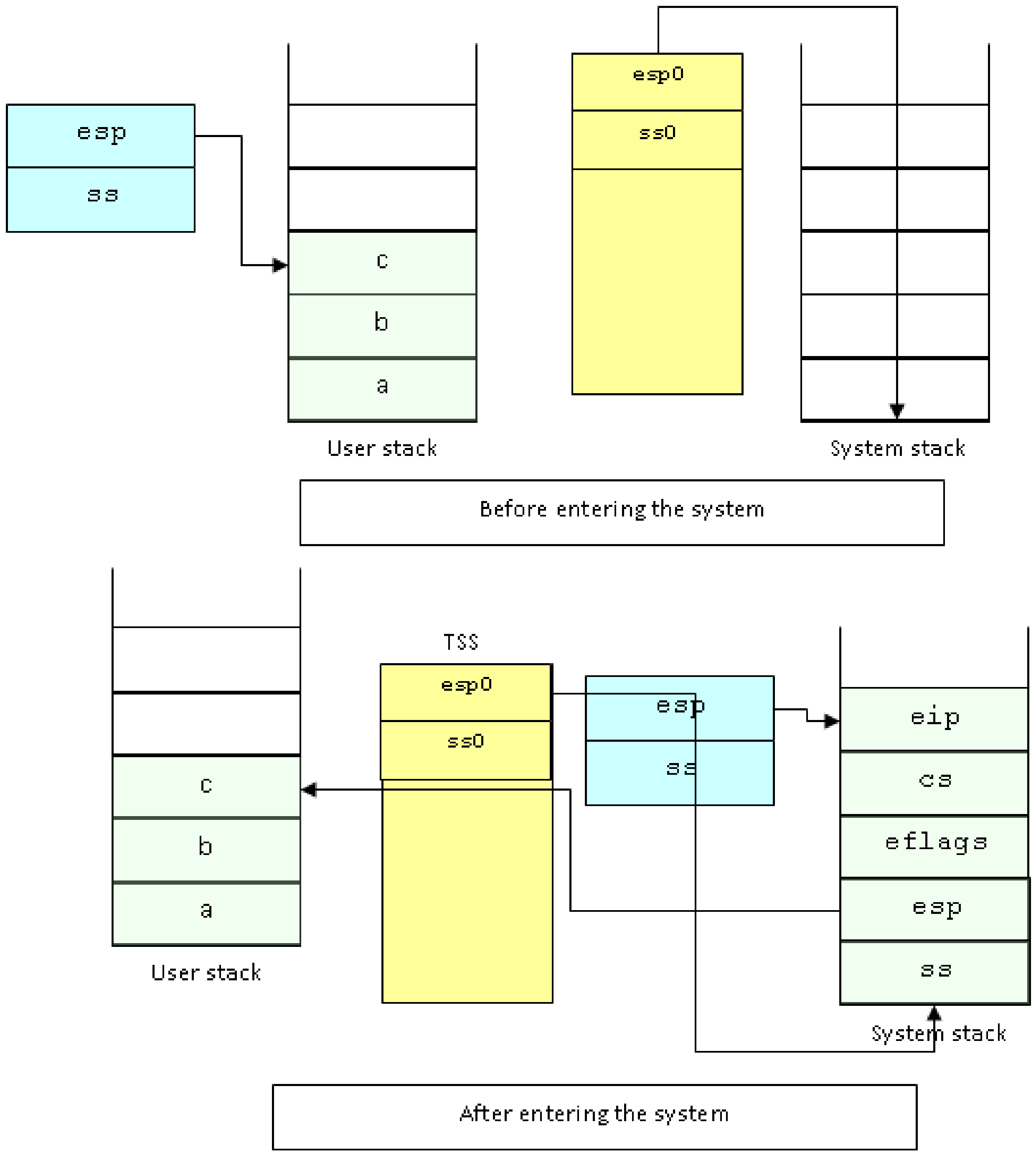


Fig. 14. The esp0 and ss0 fields of the TSS show the location of the system stack used in switch from user mode to system mode

Logical space in ZeOS

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

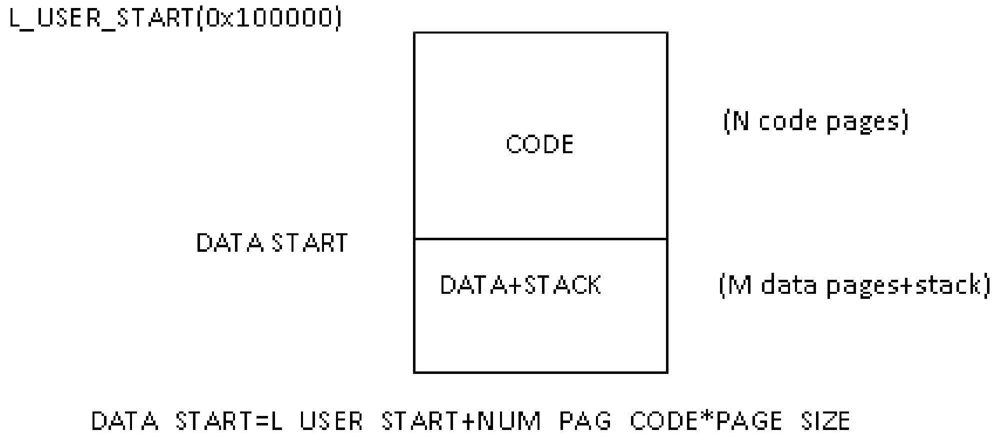


Fig. 15. Linear address space

Figure 15 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 0xffffff) correspond to the system space (see the initialization code in file mm.c). Each entry uses 4KB (it is the page size). L_USER_START»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_PAG_CODE pages for the user code and NUM_PAG_DATA pages for the data and user stack. In the initial ZeOS configuration, NUM_PAG_CODE=
and NUM_PAG_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 16).

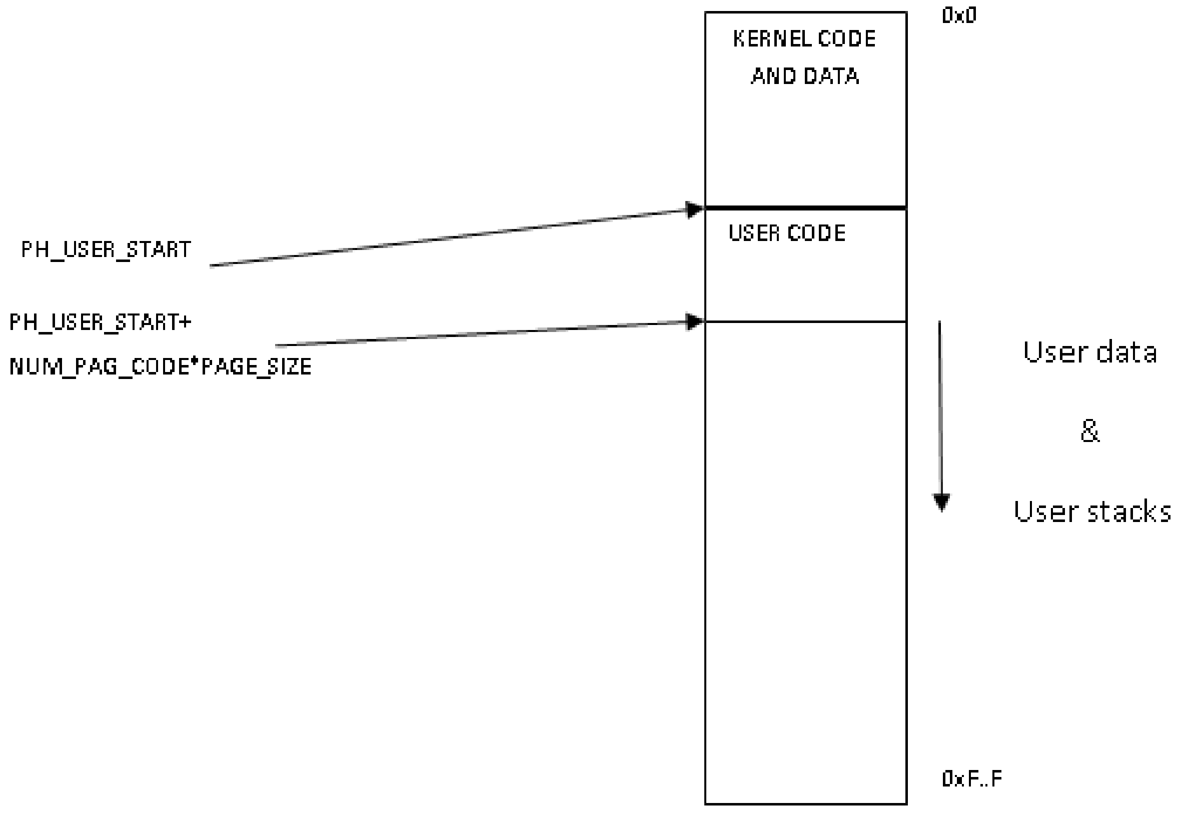


Fig. 16. Physical memory in Zeos

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»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)
```

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 17).

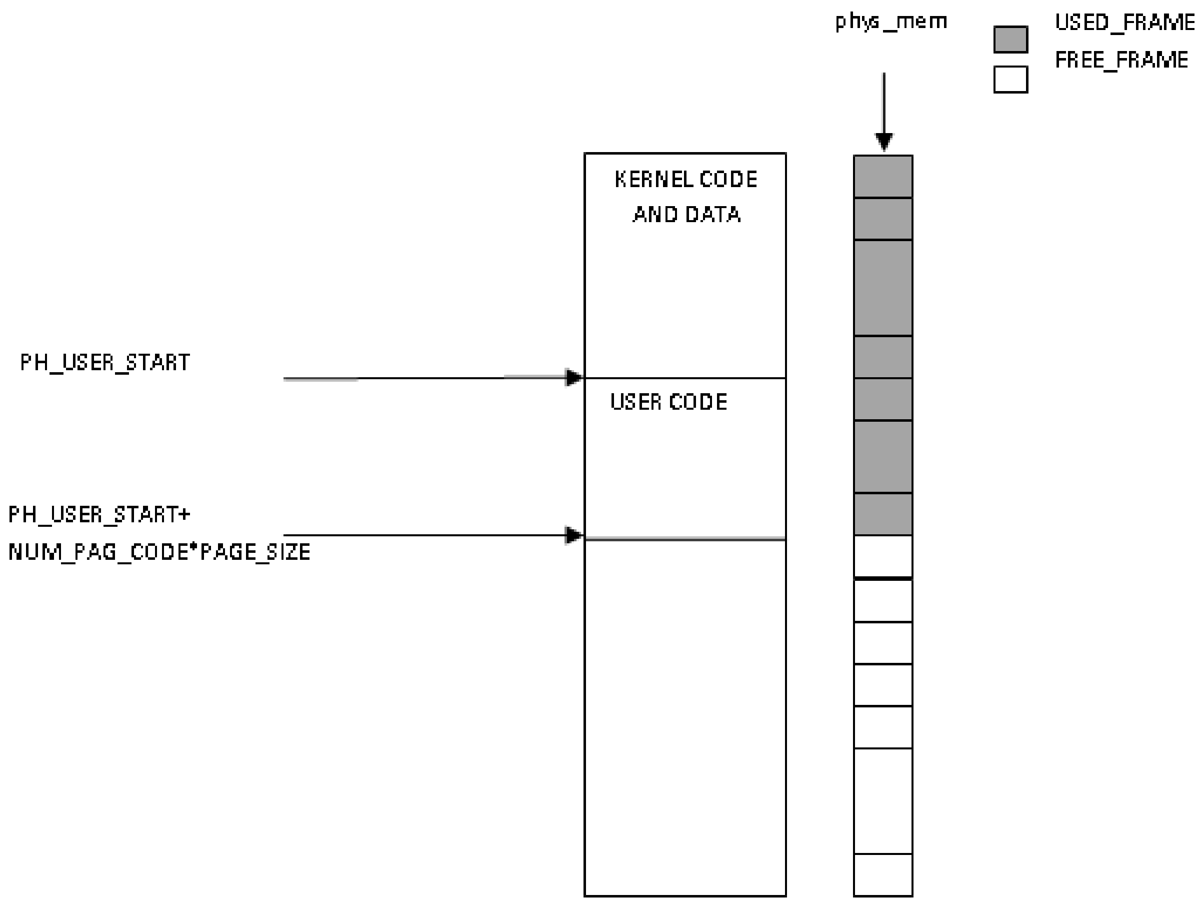


Fig. 17. Initializing physical memory and phys_mem table

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.

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 an execution context for the procees to restore it when it gets assigned the cpu (see section 4.5) and executes cpu_idle.
4) Define a global variable idle_task

   ```
   struct task_struct * idle_task;
   ```

5) 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 routine that performs the context switch 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 18). 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).

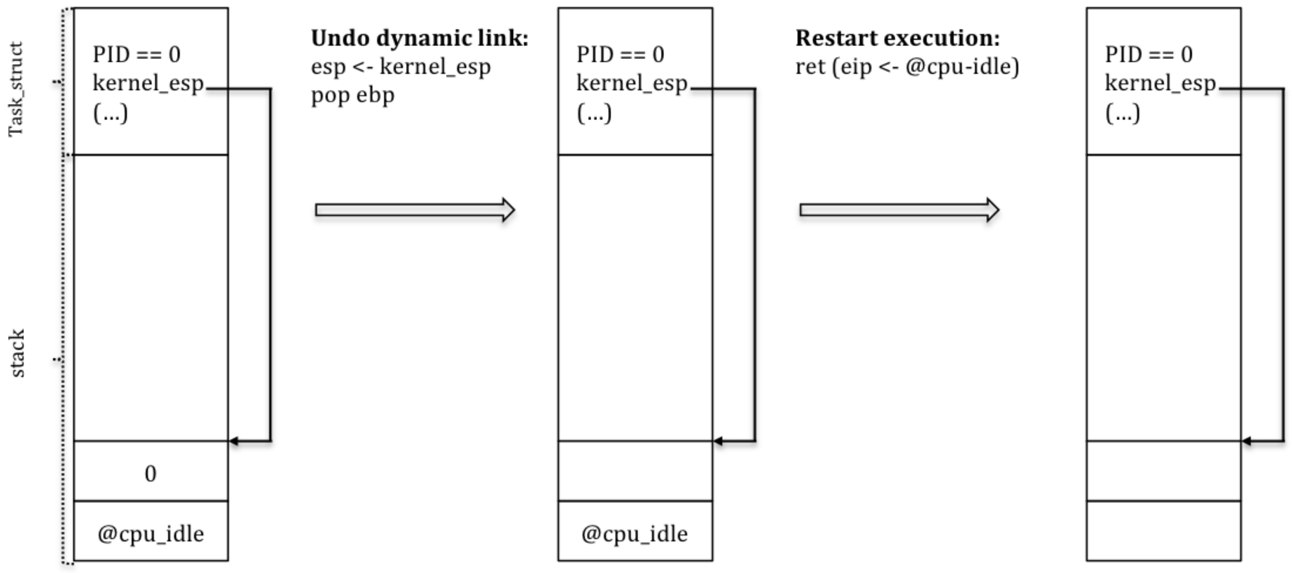


Fig. 18. Idle process: putting it on execution

4.4.2 Init process

The **init process** is the first one executed after booting the OS. Its PID is 1 and it is a user process, executing 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) 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.
3) Update the TSS to make it point to the new_task system stack.
4) 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.

4.5 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 is a wrapper that 1) saves the registers ESI, EDI and EBX[10], 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 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 19 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.

The following steps are needed to perform a context switch:

1) Update the TSS to make it point to the new_task system stack.
2) Change the user address space by updating the current page directory: use the set_cr3 funtion 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.

10. 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.

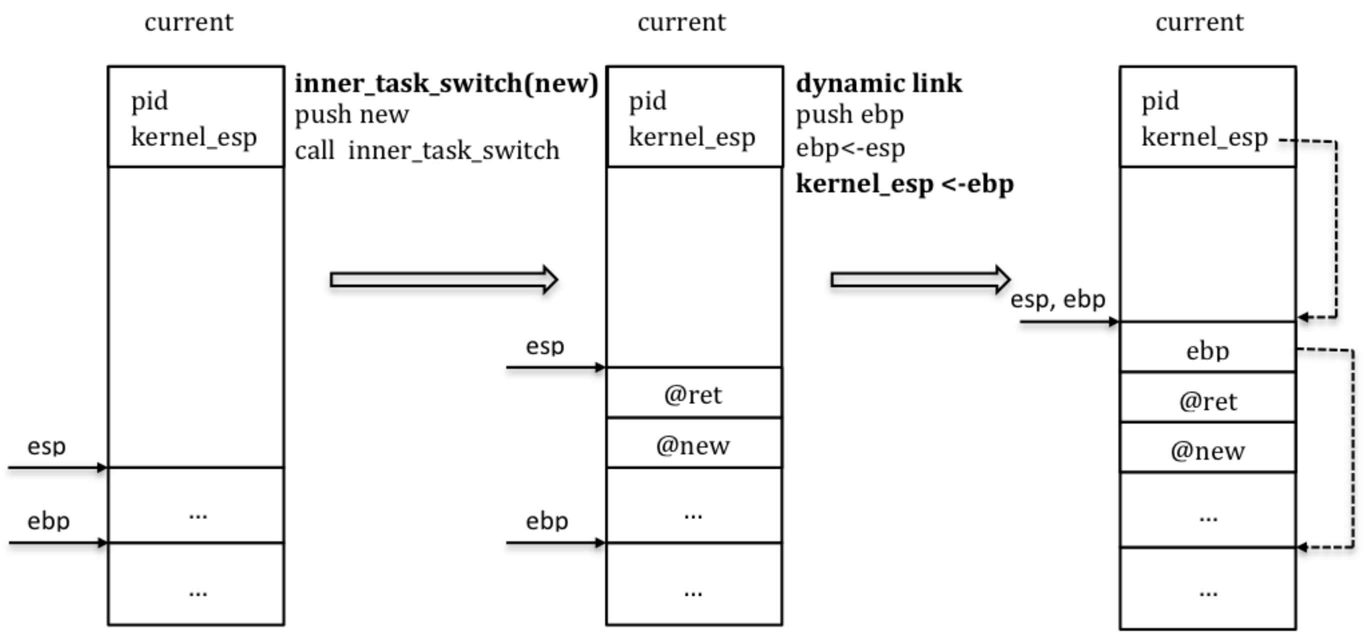


Fig. 19. Stack stages during the context switch invocation

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 in libc.c moves to eax the identifier of the getpid syscall (20), traps the kernel and returns the correspoding 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 initializated 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) 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.
    c) 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.
    d) 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)[11]:
            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.
    e) Assign a new PID to the process. The PID must be different from its position in the task_array table.
    f) 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.
    g) Prepare the child stack emulating a call to context_switch and be able to restore its context in a known position. 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

11. Remember that during the page directories initialization each task_struct is linked to one page directory, which is initialized to point to one page table (see section 4.3)

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.

h) 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.

i) 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 state of the current process and to insert it into a suitable queue (for example, the free queue or the ready queue).

```
void update_current_state_rr (struct list_head dst_queue)
dst_queue: queue according to the new state of the process
```

- 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_current_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 involves:

- Controlling how many tics the current process has spent on the CPU. Use a global variable that is decreased with every tic.
- When the quantum is over and, if there are some candidate process to use the CPU, perform a context switch. Recall that the idle process should use the CPU only if there are not any other user process that can advance with the execution.
- To perform a context switch it is necessary first to update the readyqueue by inserting the current process at the end of the readyqueue; second to extract the first process of the readyqueue, which will become the current process; and third to 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.
- Implement 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):
```

- `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.

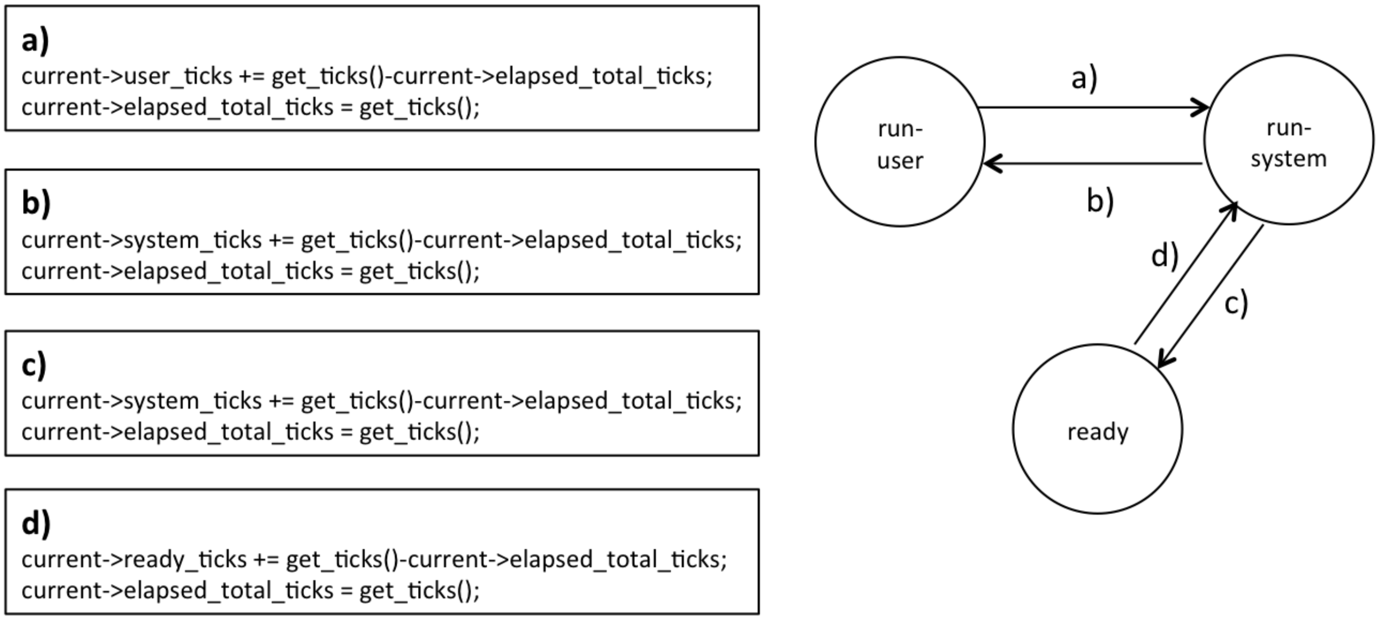
4.9 Statistical information of processes

The objective of this section is to add statistical information about processes in ZeOS. In particular, for each process, it keeps information about the time spent in user mode, the time spent in the ready queue, and the time spent in the blocked state[12]. In addition it also keeps information about the number of context switches 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 executed by the process */
        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.



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

b)
```
current->system_ticks += get_ticks()-current->elapsed_total_ticks;
current->elapsed_total_ticks = get_ticks();
```

c)
```
current->system_ticks += get_ticks()-current->elapsed_total_ticks;
current->elapsed_total_ticks = get_ticks();
```

d)
```
current->ready_ticks += get_ticks()-current->elapsed_total_ticks;
current->elapsed_total_ticks = get_ticks();
```

Fig. 20. States to consider when updating the execution time of a process

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)
```

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

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;
```

Figure 20 represents the states and transitions that we are considering at this stage of the project, and the code involved to update the statitistical 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.