AN OPERATING SYSTEM IN A NUTSHELL

Corrado Santoro

Title: “AN OPERATING SYSTEM IN A NUTSHELL”
Author: Corrado Santoro
Edition: Draft 1.2.0 - Dec 04, 2004
Copyright © 2001, 2002, 2003, 2004, 2005 Corrado Santoro (csanto@diit.unict.it), University of Catania. All rights reserved.

This book is a draft version; anyway, it contains material which are copyright by the Autor. Everything is present in this book (text, source code, figures) are copyright by the Author. None of these parts may be reproduced in form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the Author.

Contents

Chapter 1

Preface

When we planned to start writing NUXI OS, the basic problem we was called to resolve was about the real necessity to write *another* multi-tasking kernel. What we asked ourselves was "Aren't we satisfied of the huge number of existing operating systems and relevant books? Why proposing yet another OS?...." Although the bare presence of such questions could led anyone to desist in trying to write a new OS, we wanted to analyze the problem in-depth and these are the conclusion we reached.

Surely there a large number of "famous" books, like [bach][tanenbaum][silberschatz][stallings], which provide a complete explaination of OS design basics, presenting the most known techniques used to solve all the problems related to task switching, scheduling, memory management, etc.; some of these books, such as [bach], also present a full description of the functioning of internal system calls, giving their pseudo-implementation, while other books, such as [tanenbaum-minix], explains OS basics by means of the analysis of the source code of a real operating system.

However, many people are often not satisfied of these descriptions since they would like to see a *real* working implementation of the techniques onto a full-functioning operating system. Surely, they could anaylize the source code of Linux or Minix (as we made before writing this text), but this is could be too much time-expensive: You have to lose yourselves into a very large number of source code lines, include files and makefiles, and often you are unable to gain control over all the "secrets" of the analized operating system. Except Linus Torvalds, Andrew Tanenbaum and maybe few others, not so many people are able to do this and to understand the details.

Moreover, if you are a student of an operating system course, probably you would not like to spend much time in source code analyzing and, at the same time, you would understand as much as possible of the secrets of a real OS implementation. In this sense, our basic though was: "OK, we know how a task switch should be performed and how virtual memory should be implemented, but if we want to really design and implement an operting system from scratch,

what have we to do? which kind of structures have we to implement? what kind of processor instructions have we to use to handle memory space, task switching, exceptions and system calls?".

Dealing with all of these issues lead us to design **NUXI**, a very very light microkernel for Intel x86 platforms providing support for multi-tasking, memory management and device drivers. Its name means **N**ot a **U**ni**X** **I**mitation: yes! it is not another Unix/Minix/Linux clone, it is only a simple micro-kernel for educational purpose. Surely, we wrote it from scratch but we borrowed many design and implementation techniques from Minix/Linux, just to help students to understand also how the operating system they use daily works.

We reported our experience in this book, explaining each single byte of NUXI using its real souce code, which is given in the figures inside each Chapter. Moreover, since our aim is to provide a description as much comprehensive as possible, we added some sections dealing with the functioning of multi-tasking support of Intel processors, basic BIOS interrupt services, interrupt and timer circuitry, "misterious" assembler directives, etc. We will travel together in the secrets of NUXI starting from the boot stage, following protected-mode processor switching, multi-task preparation, memory handling, etc., till reaching the user-level processes. In addition, just to allow a better comprehension, the last Chapter, which is called "What's the meaning of...?", explains some minor issues/tips which are not useful to understand the core of the explained routines, but allows non-expert readers to enrich their knowledge. A diamon "♢" sign in the text means that you may find more information in the "What's the meaning of ...?" Chapter.

Even if we tried to deal with most of the issues related to kernel design, this text is not intended to replace other authoritative operating systems books! To read this text, you must be familiar with 8086 assembler, C language, OS structure basics, task scheduling, memory management techniques, and device support architectures. For this reason, it is intended to be a companion text for an operating system course.

The Authors

Part I

NUXI Explained

Chapter 2

Presenting NUXI

Let's start to know NUXI. Since we think that you would see what NUXI is able to do, our first step will be to compile and run NUXI. Subsequently, we will learn how NUXI is composed, by looking at its internal structure and analzing each module and the relevant characteristics. Finally, we will see how to write, compile and run user programs inside NUXI.

2.1 Compiling, Installing and Running NUXI

Compiling and launching NUXI is quite simple. First of all you need a Linux system (or the CygnusWin system if you have a Windows machine) provided that the "gcc" complier and the "make" tool are present in your system. To install the developing version of NUXI, copy the file "nuxi.tgz" onto your hard disk and unpack it (using the command "tar zxvf nuxi.tgz"). A directory "nuxi" will be created. Reach this directory and run first the command "./configure.sh" and then "make"; if no error occurs (we hope!), the following files will be generated:

- **nuxi.img**: the complete installation image, it contains the boot sector and the kernel image (see Chapter 5)
- **boot.lst**: the disassembled boot sector code
- **nuxi.lst**: the disassembled kernel code

Now you are ready to install your kernel onto a bootable floppy. This is done using the command "make install". At this point, if you reboot your computer using the boot disk created, NUXI will start and you will see a page like the one of Figure ?????.

After the boot, NUXI starts a simple shell, characterized by the nuxi$ prompt, which accepts the following commands:

Command	*Meaning*
`cat sys:tasks`	Prints the list of current tasks
`cat sys:mem`	Prints information about memory space
`clear`	Clear console screen
`fifo`	Tests the character FIFOs
`bfifo`	Tests the block FIFOs

While you analyze NUXI code and (possibly!) patch it, if you do not want to reboot your PC every time in order to test your implementation, you may download the open-source PC emulator "bochs" (http://bochs.sourceforge.net), or buy the VMWare^{TM} product (http://www.vmware.com). Please note that a configuration file to run NUXI using "bochs" is included in the NUXI distribution.

2.2 NUXI Basic Features

The main idea we kept in mind when designed NUXI is to build a *light*, *fast* and *real-time* kernel suitable for both educational purpose and specific applications (such as process control) running on embedded platforms. To this aim, we first designed the scheduler and some concurrency control services (the microkernel), and then we built around this all kernel services and the user layer. NUXI is thus a microkernel-based system: the kernel presents a modular structure allowing to add easily more services simply by including the source file in the source directory tree and using some microkernel hooks provided just to insert the additional services. At the same time, NUXI is monolithic, in the sense that all modules – microkernel, kernel, user library and user programs – are compiled and linked together thus creating a single binary file (which, in particular, is a boot floppy image). This is due to the fact that NUXI is a system designed to be as small as possible and (currently) does not have a file system (probably in future versions, a file system manager will be added). Due to this "small and light" requirement, user programs in NUXI do not behave as *processes*, but there is a *single user process* which can run *multiple threads*. In addition, NUXI does not provide virtual memory management, because NUXI is not intended to be run on workstations or servers like other more authoritative OS, but it is designed to be employed in specific application fields (such as small computing environment) which do not need more than the RAM which could be installed on a PC platform.

Finally, the last characteristic we kept in mind was the *standardization*: system calls and utility functions provided by the NUXI user library are designed in order to be conform to ANSI-C and POSIX.

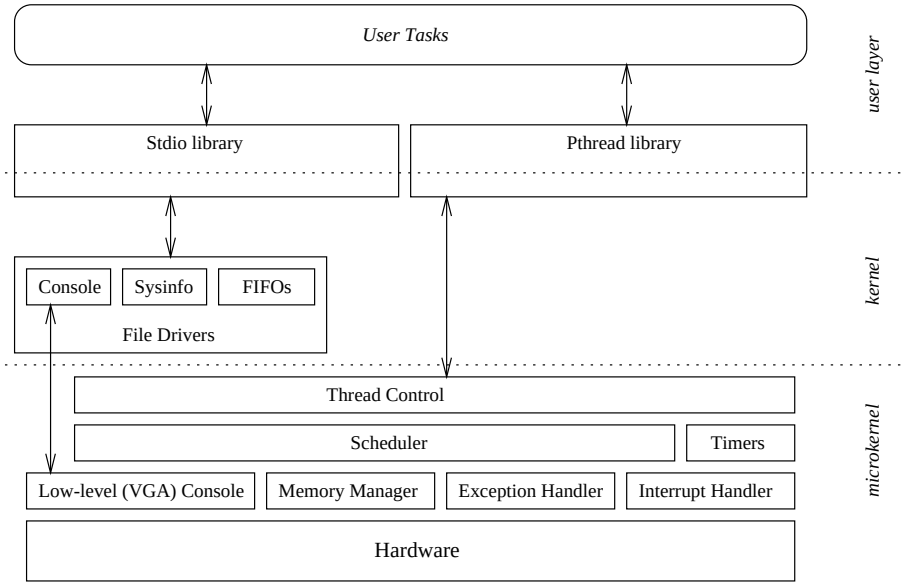

Figure 2.1: NUXI Modular Structure

2.3 NUXI Structure

The modular composition of NUXI is depicted in Figure 2.1 which shows that basically NUXI is composed of the following three layers:

MicroKernel. This layer is very thin and includes the modules for tasks and CPU management. The microkernel provides all services for interfacing the kernel layer with the CPU. In particular, the **low-level console**, handles VGA output by offering a set of calls to print characters and strings onto the screen, perform scrolling functions and handling cursor positioning. The **memory manager** handles memory heap, offering services to allocate and free memory blocks. The **interrupt handler** and the **exception handler** allow instead to set and remove interrupt-service routines for processor exceptions, software interrupts, and hardware IRQs. Upon the interrupt handler runs the **timer module**, entailed to handle flow of time and software timer, and the **scheduler**, one of the main part of NUXI. It is a standard round-robin preemptive scheduler with a priority management mechanism able to handle both *dynamic* and *static* priorities: dynamic priorities are used for non-realtime processes, and are managed on the basis of the most recent CPU usage of a process (thus implementing a UNIX-like fair scheduler); static priorities are reserved for realtime processes and are fixed on the basis of time urgency of each process. The scheduler is managed by the **thread control module**, responsible of handling process creation and destruction, and concurrency control. The latter function is performed through *semaphores*, *condition variables* and *wait channels* which are offered to the higher levels of kernel running upon this module. The presence of a thread control placed at a low level

and running directly upon the scheduler allows to write a *pre-emptible kernel*, which is a mandatory characteristic when you develop a realtime OS. Other OSs, like Unix/Linux, do not allow pre-emption in kernel mode: this introduces impredictable latencies which impede hard-realtime processes to meet the requested time constraints. NUXI, instead, is developed using some techniques which allows to pre-empt *everything*, both in kernel and user mode. Kernel routines make a wide use of semaphores for critical sections (instead of clearing interrupts) and each process has both a user-level and kernel-level context in order to allow pre-emption in kernel code when a process issues a system call.

Kernel. The kernel layer is basically composed of the **file driver module** and the **system call handler**. The **file driver** manages installable drivers which offer filing services; in particular it provides an abstraction layer handling all filing system calls (open, read, write, close, ioctl, etc.) and its functioning is similar to that of character device driver management module of the Unix OS. Inside this module three file drivers run: the **console driver**, handling screen and keyboard I/O, the **sysinfo driver**, a read-only driver providing system information such as process list or memory statistic data, and the **FIFO driver**, which offers inter-process communication services.

User. At the highest level user programs and user libraries are placed. In particular the **system library** implements ANSI-C standard I/O and POSIX-thread services, providing the needed interface between the kernel and user processes.

2.4 NUXI Source Organization

2.5 Writing Your Programs in NUXI

In this Section, we will explain how to write user programs in NUXI. If you want to write kernel parts or patch it, please read the Part II (Nuxi Revealed) of this book.

To write a user program in NUXI is quite simple. Since the kernel, once it finishes all its startup procedures, calls the function `void user_main(void)`, you have to add a source file which implements your program as this function and modify the Makefile by setting the USER variable to the name of your source file (without the ".c" suffix). As an example, you may see the file "userproc.c", which implements a very simple shell for testing purpose. You can write your program using a subset of the standard ANSI-C and POSIX library calls, which are described in Section 2.6, and the kernel services described in Section 3.

2.6 NUXI C Library

This Section reports the prototype of the library functions currently implemented in NUXI.

2.6.1 Standard I/O

Include files:

- `stdio.h`

Available functions:

- `int vfprintf(FILE *f,char *fmt,va_list ap);`
- `int printf(char *fmt,...);`
- `int fprintf(FILE *f,char *fmt,...);`
- `int fprintf(FILE *,char *fmt,...);`
- `int fputc(int c,FILE * f);`
- `int fputs(const char *s,FILE * f);`
- `int putc(char c,FILE * f);`
- `int putchar(char c);`
- `int fgetc(FILE * f);`
- `char * fgets(char * s,int size,FILE * f);`
- `char getchar(void);`
- `int vsprintf(char * s,char *fmt,va_list ap);`
- `int sprintf(char * s,char *fmt,...);`

2.6.2 Standard lib

Include files:

- `stdlib.h`

Available functions:

- `long int strtol(const char *nptr,char **endptr,int base);`
- `int atoi(char * s);`

2.6.3 String Library

Include files:

- `string.h`

Available functions:

- `int strlen(char * s);`
- `char * strcpy(char * dst,char * src);`
- `char * strcat(char * dst,char * src);`
- `char * strncpy(char * dst,char * src,int len);`
- `int strcmp(char * s1,char *s2);`
- `int strncmp(char * s1,char *s2,int n);`
- `char * strchr(char * s,char c);`
- `void * memcpy(void * dest,void * src,int size);`
- `void * memset(void * s,int c,int size);`

2.6.4 C-Type Library

Include files:

- `ctype.h`

Available functions:

- `int isspace(int c);`
- `int isdigit(int c);`
- `int isalpha(int c);`
- `int toupper(int c);`
- `int tolower(int c);`

2.6.5 File Operations

Include files:

- `fcntl.h`

Available functions:

- `int open(char * name,int mode);`
- `long write(int fid,char * buf,long size);`
- `long read(int fid,char * buf,long size);`
- `long ioctl(int fid,int cmd,void * arg);`
- `int close(int fid);`

Chapter 3

NUXI Kernel Services

Part II

NUXI Revealed

Chapter 4

Intel x86 Family Basics

[TBD]

Chapter 5

The Startup Process

Let's start dealing with NUXI design and implementation from the boot process. NUXI is currently designed to boot only from a floppy, because handling the OS loading process from a floppy is more simple than using an hard disk, since we do not need to deal with partition tables. At the same time, we think that, when you will perform some experiments by modifying kernel source, you will prefere to avoid any data loss, on your hard disk, deriving from a possible bug in the OS install process.

Once NUXI is compiled, the "makefile" produces a floppy image binary file which has the following structure:

1. The first 512 bytes compose the **boot sector**. It is loaded and executed by the BIOS when the computer is started-up.

2. The remaining bytes compose the **OS kernel image**, which will be loaded and executed by the code present in the boot sector.

When this image is copied onto a floppy, the boot sector is placed in the first sector of the disk (track 0, head 0, sector 1), and the OS kernel image is placed in the subsequent sectors.

5.1 The Boot Sector

When your PC is started-up, the ROM BIOS loads the first sector of the floppy at memory address 07C0H:0000H (linear 07C00H)[1] and then performs a long jump to that location. The boot code has to load the kernel into memory and then to run it. To this aim, several issues have to be taken account. First of all, we must choose the memory location onto which starting to load our kernel. In order to do not waste memory space (even if we have a huge amount of RAM!), we should place the code as near as possible to the beginning of our memory. As Figure 5.1 shows, which

[1] Please note that, at start-up, the processor is in real mode and thus its addressing mode behaves like 8086 CPU.

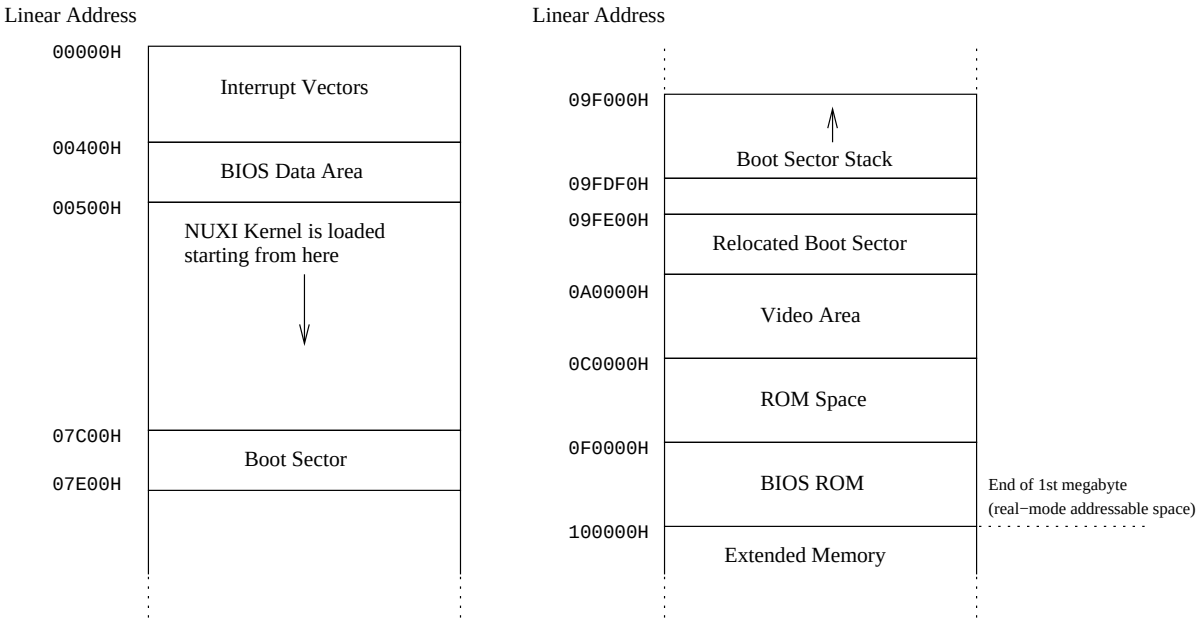

Figure 5.1: PC Memory Map

reports the memory map of standard PC, locations from 0000:0000 to 0000:0400H are occupied by the interrupt vectors (4 bytes, seg:off, for each interrupt vector times 256 interrupts), while the memory from 0000:0400H to 0000:0500H is used by the BIOS to store its data (the so-called "BIOS Data Area"). Thus, since we will use BIOS services in this boot process (for example the "read sectors" service for kernel loading) we must not overwrite data stored in these locations. The lowest memory location available is therefore 0000:0500H therefore the boot code will load the kernel image placing it starting from this address. However, in performing kernel loading, we must do care to code size: if it is greater than 30464 bytes (7700H), location 7C00H (500H + 7700H) is reached, thus overwriting the boot code which is still running. To solve this problem, which is common for all existing operating systems, the boot code generally relocates itself to a higher memory area before performing any other operation. We chose the highest memory area available in the first megabyte of RAM which, according to Figure 5.1, is location 09FE0H:0000H (linear 09FE00H)[2]. The final choice we have to make is establish the memory area for the stack of the boot code which, following the same guidelines, is placed before relocated boot code at linear address 09FDF0 (which corresponds to 09F00:0DF0H) $\diamondsuit$. Now we are ready to load the kernel. The complete listing of the boot code is given in Section 5.1.1 and performs the following steps:

Lines 17–18. Perform a short jump to location `next`**.** This is required to skip the `sector_count` variable definition, which is initially set to 0, and the local variable area (lines 20–23). As

[2]RAM size in the first megabyte is 0A0000H, boot sector is 200H (512) bytes long, thus 0A0000 - 200H = 09FE00H.

we will see in the following, the `sector_count` variable holds the number of sectors to be read in order to load the kernel image. Its value depends on the size of the kernel and thus it is computed at kernel compile time: once the final kernel image is produced by the compiling process, the `utils/kernelsize` utility determines the number of sectors which the kernel will occupy and stores the value directly at offset 2 of the boot sector image (the short jump instruction is two bytes long). The presence of a short jump as the first instruction of a boot sector is common technique and allows a parameter table, needed by the boot process, to be placed at a known address (location 2) in order to be filled in a second time.

Lines 25–36. Relocate boot sector. According to what we said above, the boot sector relocate itself from address 07C0H:0000H to segment BOOTSEG which is set to 09FE0H (line 10). This is done by using the "move string word" – `movsw` – opcode repeated by 256. Then a long jump to the relocated boot sector is performed.

Lines 37–47. Setup Stack and Segment Registers. First we load registers SS and SP, then, since we have some local variable in the boot sector, we need to access them. For this reason we set DS register to point to segment BOOTSEG, which contains the relocated boot sector.

Lines 48–83. Load the kernel from disk. Now we are ready to perform kernel loading. To this aim, we have to read `sector_count` sectors from disk starting from sector 2 of track 0, head 0. Please note that, even if track and head numbers start from 0, sector numbers start from 1, thus our boot sector is track 0, head 0, sector 1 and the kernel image starts from track 0, head 0, sector 2. To read floppy sectors, we use the "read sectors" service of the Disk BIOS, by calling INT 13H $\diamondsuit$ with AH=2 (Figure 5.2 reports the meaning of the registers for this service call). This service is able to read multiple sectors but, in some BIOS, it cannot go beyond a track. For example, if want to read 20 sectors of a 1.44 MBytes floppy (which has 18 sectors per track) with a single call, the BIOS service will fail, since the requested sectors do not surely belong to the same track. For this reason, we do not read the whole kernel image using a single service call but we invoke the BIOS "read sector" service for each single sector until we reach `sector_count`. To this aim, we use the varaibles `sector`, `track` and `head` to store respectively the sector number, track number and head number of the disk sector to be read, and `address` to store the memory offset address at which the sector will be loaded. As the source code reports, the latter variable is initialized to `kaddress` which is defined as 0500H in file "segments.s". The loop which reads kernel sectors is quite simple: we prepare all registers to read one sector, print an 'E' and retry if an error occurs, if read succeeds increase memory location by 512 (each sector is 512 bytes long), increase sector number and, if needed, change head and/or track number. The loop counter is register CX which is initialized to `sector_count`.

Lines 85–87. Turn motor drive off. If no error occurs, we turn the drive motor off by out'ing the value 0 to port 03F2H.

Register	*Meaning*
AH	2 (“read sector” service)
AL	Number of sectors to read
ES:BX	Memory address at which the loaded sectors have to be places (SEG:OFS)
CL	First sector number to be read
CH	First track number to be read
DH	First head number to be read
DL	Drive number (drive A=0, drive B=1, drive C=2, etc.)

Figure 5.2: INT 13H Registers Meaning

Line 91. Start the kernel. Finally we start the loaded code by performing a long jump to location 0000:0500H.

The last two lines of the boot sector souce places the bytes 0x55 0xAA at the end of the sector itself (offset 510). This is the “boot signature” needed to signal the BIOS that the sector contains a boot code.

5.1.1 Boot Sector Source Code

```
1:  #
2:  # NUXI Boot Code
3:  # Copyright (C) 2001 Corrado Santoro <csanto@diit.unict.it>
4:  #
5:  # This code is loaded from BIOS at address 0000:7C00
6:  #
7:
8:  .include "segments.s"
9:
10:  BOOTSEG = 0x9FE0
11:  STACKSEG = 0x9F00
12:
13:  .global boot
14:
15:          .code16
16:          .balign      0x08
17:  boot:
18:          jmp next
19:  sector_count: .word  0
20:  sector: .byte        2
21:  head:   .byte        0
22:  track:  .byte        0
23:  address: .word        kaddress
24:
25:  next:
26:                                    # now move boot sector at BOOTSEG:0000H
```

```
27:          movw    $0x7c0,%ax        # boot sector is loaded at 07C0H:0000H
28:          movw    %ax,%ds           # thus load DS to access this area
29:          movw    $0,%si            # ds:si = loaded boot sector area
30:          movw    $BOOTSEG,%ax
31:          movw    %ax,%es
32:          movw    $0,%di            # es:di = destination area
33:          movw    $256,%cx          # word count
34:          cld
35:          rep     movsw             # relocate sector
36:          ljmp    $BOOTSEG,$go      # go to relocated code
37:  go:
38:          movw    $STACKSEG,%ax
39:          movw    %ax,%ss
40:          movw    $0x0df0,%sp  #place stack at STACKSEG:0DF0H (before boot sector)
41:          call    writedot          # put a dot onto the screen to signal
42:                                    # the first boot stage
43:
44:          movw    $BOOTSEG,%ax      # now boot sector is at BOOTSEG:0000H
45:          movw    %ax,%ds           # thus load DS to access this area
46:          movw    sector_count,%cx  # loop for the number of sectors
47:                                    # to read
48:  do_read:
49:          pushw   %cx
50:          movw    $ksegment, %ax    # prepare register for sector read
51:          movw    %ax, %es          # address (segment)
52:          movw    address, %bx      # address (offset)
53:          movb    $0x02, %ah        # service 2 (read sectors)
54:          movb    $0x01,%al         # sector count=1 sector
55:          movb    sector, %cl       # sector number
56:          movb    track,%ch         # track number
57:          movb    head,%dh          # head number
58:          movb    $0x0,%dl          # drive number (0=A:)
59:          int     $0x13             # read a sector of microkernel image
60:          jnc     read_ok
61:
62:          movb    $0x45,%al         # if error, display 'E' and retry
63:          call    writechar
64:          popw    %cx
65:          jmp     do_read
66:
67:  read_ok:
68:          call    writedot          # sector loaded successfully
69:                                    # now go to next sector
70:          addw    $512,address      # sector size if 512 bytes
71:          incb    sector
72:          cmpb    $19,sector         # are we at the end of track?
73:                                     # (1.44 MB floppy => 18 secs./track)
74:          jne     read_next
75:          movb    $1,sector
```

```
76:          incb    head               # end of track reached => next disk face
77:          cmpb    $2,head            # did we read both faces ?
78:          jne     read_next
79:          movb    $0,head            # both faces read, go to next track
80:          incb    track
81:  read_next:
82:          popw    %cx
83:          loop    do_read            # loop to total number of sector to read
84:
85:          movw    $0x03F2, %dx       # turn off drive motor
86:          xorb    %al, %al
87:          outb    %al, %dx
88:
89:          call    writedot           # motor off successfully
90:
91:          ljmp    $ksegment, $kaddress
92:
93:  # write a dot into the screen
94:  writedot:
95:          movb    $0x2e,%al
96:  writechar:
97:          movb    $0x0e,%ah
98:          movw    $0x0007,%bx
99:          int     $0x10
100:          ret
101:
102:          .org   510, 0
103:          .byte 0x55,0xAA
```

5.2 Switching into Protected Mode

The first step performed by the loaded kernel is to switch the processor in protected mode. This is performed by the NUXI startup code present in file `start.s` and reported in Section 5.2.1. First, we set the DS register equal to CS since our data is placed in the code segment (lines 22–23). The next instruction clears the processor interrupt flag to impede the CPU to respond to any incoming external interrupt, operation required since we are going to change processor addressing mode; in fact, until the interrupt service routines which use protected mode addressing will be not properly installed, an interrupt occurring during this initialization phase, if handled, could hang up the system. At this point, we load the GDT register of the CPU (line 26) with the data stored at location `gdtr` (located at lines 81–92): it contains the size and the linear address of our global descriptor table. The GDT size (in bytes) is computed using the constant `gdt_entries` counting the number of entries of our table, while the GDT starting address is referred by label `gdt` whose linear address is resolved by the linker (see below).

Let's show how our GDT is organized. As we can see in the source code of Section 5.2.1, each entry is compose of four words, according to Intel specifications (see Chapter 4). The first entry is the "null entry" and correspond to selector 0; the entry is filled to zero just to cause a "segment not-present" [general protection fault??] exception (P bit is 0) when the program references a NULL pointer. This is a common design technique for operating systems and avoids dirting memory when a program tries to use a non-initialized pointer (a common bug when you develop software using C language); instead, an exception is raised to allow the kernel to stop the program which caused incorrect addressing and, possibily, to print a proper message on the screen (aren't you familiar with "Segmentation Fault" messages?). The next two entries (selectors 08H and 10H) will refer respectively to code segment and data segment in our kernel. Both descriptors map the entire addressable memory: a 4 GBytes segment starting from the beginning of memory (address 0). As Figure 5.3 shows, by setting the segment limit to the maximum and the Granularity bit to 1 (the limit part of selector thus refers to 4 KBytes pages) we are able to access the whole memory from a single segment in order to map each location of the segment to the numerically correspondent physical location. For example, accessing memory address 0FF0100H using selector 08H will be equivalent to access memory at linear address 0FF0100H. Both segments are thus relevant to the same memory space, but the former (relevant to selector 08H) is *executable*, i.e. the bit E is set, while the latter (relevant to selector 10H) is a data segment. Choosing this virtual-to-physical correspondence is mandatory since the linker resolves symbols considering a contiguous addressing space, thus mapping data and code segments on the overall 4 GBytes addressing space avoids the need for run-time relocation.

Indeed, the global descriptor table contains other segments, but they are left un-initialized since will be used to map Task State Segments of the various tasks which will be launched later (see Chapter 9).

After loading the GDT register (and after writing another dot!), we are ready to switch into protected mode, operation performed by setting the bit 0 of the processor's machine status word (MSW) also called register CR0 (lines 30–32). However, switching is performed by the processor only after an inter-segment long jump. Thus, following the suggestion of Intel software developers' manual [rif.], we perform a simple short jump in order to flush the processor's instruction cache (lines 34–35), and then a long jump to address `k_start` of selector 08H (lines 37–40). This long jump is forced to work in 32-bit addressing mode by using the 32-bit addressing override instruction. Since we do not know whether the compiler treats a long jump using 16 or 32 bit addressing (we are still in the .code16 part of the code), we place directly in the source code the requred op-codes instead of using the `jmpl` mnemonic: op-code 66H forces the processor to treat the next instruction with 32 bit addressing, op-code 0EAH is the long jump instruction, the following long word is the offset of the target of the jump and the last word is the selector (`code` constant is set to 08H in the "segments.s" source file).

Program continues execution at location `k_start` (line 55 – we are now in the .code32 section since here the execution is in protected mode), here we load all the segment selectors registers

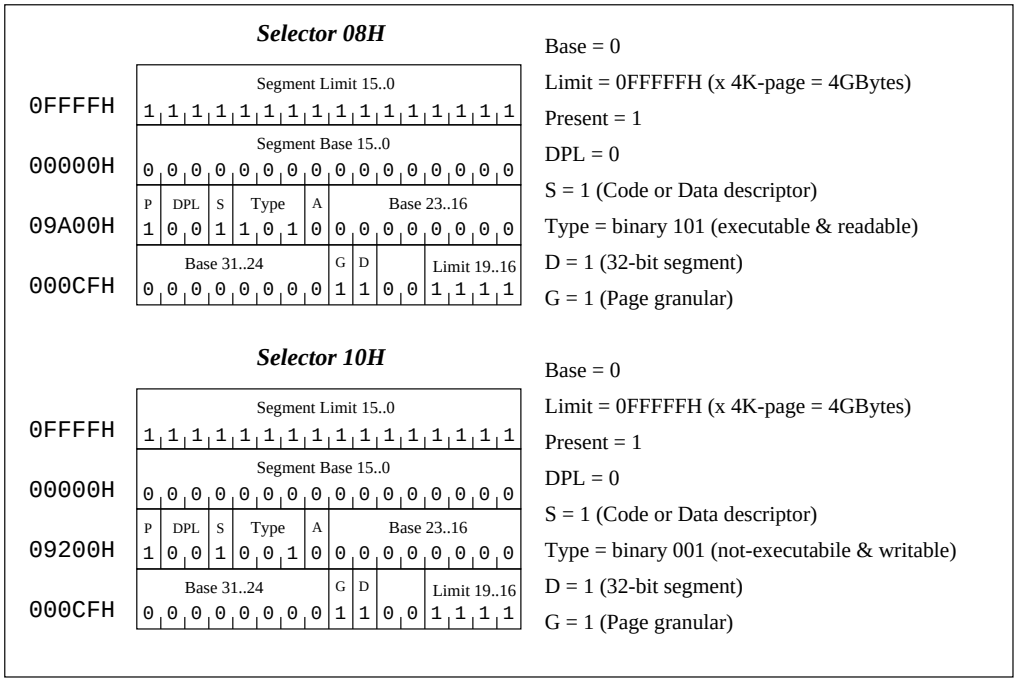

Figure 5.3: The Segments used by the Kernel

of the processor (except CS) in order to point to the data segment 10H (`data` constant is set to 10H in the "segments.s" source file). Then, we have to setup a temprary stack (which will be freed when the first task will be started) by setting ESP register to the highest memory location in the first megabyte, i.e. location 09FFF0H. Finally we jump to location `kernel_main` which refers to the kernel startup routine, written in C, and present in the `kernel.c` source file.

5.2.1 Kernel Startup Source Code

```
1:  #
2:  # NUXI Startup Code
3:  # Copyright (C) 2001 Corrado Santoro <csanto@diit.unict.it>
4:  #
5:
6:  .include "segments.s"
7:
8:  .global _gdt
9:  .global start
10:  .global _start
11:  .global _load_idtr
12:  .global _dummy_iret
13:
14:          .code16
```

```
15:
16:
17:    .balign  0x08
18:  start:
19:  _start:
20:          call    writedot  # jump to start code successfully
21:
22:          movw    %cs,%ax    # setup DS equal to CS
23:          movw    %ax,%ds
24:
25:          cli                 # disable interrupts
26:          lgdt    gdtr    # load global descriptor table
27:
28:          call    writedot  # gdt setup OK
29:
30:  # OK. Set Bit0 of Machine Status Word -> Set Processor in Protected Mode
31:          movw    $0x1,%ax
32:          lmsw    %ax
33:  # force flushing of instruction cache
34:          jmp     flush
35:  flush:
36:  # NOW IN PROTECTED MODE. Perform long jump to kernel startup code
37:  #       jmpi    0x10,pm
38:          .byte   0x66,0xea
39:          .long   k_start
40:          .word   code
41:
42:  # --------------------
43:  # subroutines for 16bit mode
44:          .balign 0x08
45:  # write a dot into the screen
46:  writedot:
47:          movb    $0x2e,%al
48:  writech:
49:          movb    $0x0e,%ah
50:          movw    $0x0007,%bx
51:          int     $0x10
52:          ret
53:
54:          .code32
55:  k_start:
56:          movw    $data, %ax
57:          movw    %ax, %ds
58:          movw    %ax, %es
59:          movw    %ax, %fs
60:          movw    %ax, %gs
61:          movw    %ax, %ss
62:          movl    $0x09fff0,%eax
63:          movl    %eax,%esp
```

```
64:
65:  # booting the kernel main function (in kernel.c)
66:          jmp     kernel_main
67:
68:          .set    gdt_size,16
69:
70:          .balign  0x08
71:          .word  0x0000
72:  gdtr:   .word  (gdt_size * 8)-1
73:          .long  gdt
74:
75:          .balign  0x08
76:  gdt:
77:          .word  0x0000, 0x0000, 0x0000, 0x0000
78:          .word  0xFFFF, 0x0000, 0x9A00, 0x00CF
79:          .word  0xFFFF, 0x0000, 0x9200, 0x00CF
80:          .rept gdt_size-3
81:          .word  0x0000, 0x0000, 0x0000, 0x0000
82:          .endr
83:
84:          .data
85:  _gdt:
86:          .long gdt
87:          .end
```

5.3 Preparing the Kernel

Now we can switch to source file `kernel.c` (whose listing is given in Section 5.7) which is (finally!) written in C and take a look of function `kernel_main` (lines 137-161). This function contains several calls to initialization functions and ends by launching the first kernel task. We do not detail here these routines since they belong to modules which will be described each in its relevant Chapter. In particular console management, interrupts handling and task switching will be dealt with in the above Chapter, while the next section of this Chapter will describe A20 gate enabling, memory testing, and co-processor resetting.

The `kernel_main` function ends with calling `task_add` which launches the routine passed as parameter as a new task. If the task is the first task launched, `task_add` must never return, but if a return occurs maybe there is a bug in the kernel; in this case, we disable scheduler by clearing the interrupt bit, print a panic message on the screen and halt the machine. We will deal with operation performed by `task_add` in Chapter 9.

5.4 A20 Gate Management

The first thing we must do (after writing the NUXI welcome message!) is to enable A20 line. What is the meaning of "A20 enabling"? Well, A20 is the line 20 of the memory address bus; in PC platforms (since the IBM PC-AT), this line is forced to 0, at computer startup, by a suitable circuitry in order to allow 80286 and subsequent processor, when operating in real-mode, to behave exactly like 8086/8088. Let us explain this in a deeper way. When you access memory in an 8086/8088 processor, the addressing unit performs the calculation `segment*16+offset` and then reads/writes memory using the resulting linear address. If you try to access memory at address 0FFFFH:0FFFFH the resulting linear address will be 10FFEFH; since 8086/8088 have only 20 address lines, the above linear address is truncated into 0FFEFH. On the contrary, 80286 CPUs (and subsequent) have *more* than 20 address lines, thus they do not perform MSB truncation. This could provoke a different behavior of programs running in a 80286 processor (in real mode) but designed for 8086/8088. For this reason, designers of the "good old" IBM PC-AT added a circuitry which forces A20 line to zero. Fortunally, this circuitry can be disabled in order to allow correct addressing when the processor works in protected mode. This is what we have to do!

The code which enables A20 line is the function `a20_enable` – lines 44–57 of source code in Section 5.7. In the first AT PC, the circuit controlling this function was enabled and disabled by using a spare bit of the keyboard controller. For this reason, the first two port operations of the `a20_enable` function are relevant to port 064H and 060H (the keyboard controller, see Chapter [keyboard]!). When PS/2 PCs were introduced, designers added a different I/O port dedicated to A20 management. For this reason, in our source code, we also reported enabling of A20 by setting bit 1 of port 092H.

The `__SLOW_DOWN_IO` macros, defined in the include file "io.h" and used in A20 enabling, introduce a small delay in order to permit driven circuits to respond. This macro simply "wastes time" by performing two short jump instructions.

5.5 Testing Memory

The function `memory_test` is quite simple (lines 25–42). It aims to set the `machine_memory_size` variable to the highest available location of the installed memory. We do this by performing a write/read test on the first location of each megabyte of our memory. The local variable `memaddr` is initially set to point to the second megabyte of the memory (we assume that you have at least 1 MByte of memory istalled!!) and increased each time by 1 MByte. The test is performed by writing the word 1234H in the location pointed by `memaddr` and reading it back; the loop ends when the value read differs from 1234H or when we reached the end of the addressable memory (4 GBytes, but we think that no computer with this amount of memory exists... probably in the future... However, even if we did not tested our function with this amount of memory, it should

work correctly!). The `memaddr` variable is declared as volatile otherwise compiler optimization will skip the test assuming that it is always false. In fact, we write the value 1234H onto `*memaddr` and then we test it against the same value, thus the compiler thinks that, between the assignment and the test, no one will change the written data (uhm, a stupid compiler? No, obviously it does not suppose that we are trying to address non-exisiting memory!).

5.6 Resetting co-processor

Finally, we perform a co-processor reset. Co-processor is driven by using I/O ports 0F0H and 0F1H and its reset is simply performed by out'ing zero on these ports.

5.7 Kernel Startup Source Code

```
1:  /*
2:   * kernel.c
3:   * Copyright (C) 2001, 2002 Corrado Santoro (csanto@diit.unict.it)
4:   */
5:
6:  #include <kernel/io.h>
7:  #include <kernel/asm.h>
8:  #include <kernel/console.h>
9:  #include <kernel/interrupt.h>
10:  #include <kernel/timer.h>
11:  #include <kernel/task.h>
12:  #include <kernel/kprintf.h>
13:  #include <kernel/mm.h>
14:  #include <kernel/wait.h>
15:  #include <kernel/exception.h>
16:  //#include <kernel/mailbox.h>
17:  #include <kernel/driver.h>
18:  #include <kernel/usertable.h>
19:  #include <user/pthread.h>
20:
21:  #define KERNEL_RELEASE  "0.0.9-2"
22:
23:  void * machine_memory_size = 0;
24:
25:  void memory_test(void)
26:  {
27:    // Start memtest from 0x100000 (1 MB)
28:    // Declare as 'volatile' otherwise assignment and comparison with 0x1234
29:    // does not work due to compiler optimization
30:    volatile uint16 * memaddr = (uint16 *)0x100000;
31:    while (memaddr != 0x0) {
```

```
32:       uint32 l_addr = (uint32)memaddr;
33:       l_addr >>= 20;
34:       kprintf("\rTesting Memory: %d MBytes",(int)l_addr);
35:       *memaddr = 0x1234;
36:       if (*memaddr != 0x1234)
37:         break;
38:       memaddr += 0x80000; // goto next megabyte
39:     }
40:     kprintf(" Found OK\r\n");
41:     machine_memory_size = (void *)memaddr;
42:   }
43:
44:   void a20_enable(void)
45:   {
46:     kprintf ("Enabling A20...");
47:     outb(0xd1,0x64);
48:     __SLOW_DOWN_IO;
49:     outb(0xdf,0x60);
50:     __SLOW_DOWN_IO;
51:     outb(inb(0x92) | 0x02,0x92); // fast "A20" version
52:     __SLOW_DOWN_IO;
53:     __SLOW_DOWN_IO;
54:     __SLOW_DOWN_IO;
55:     __SLOW_DOWN_IO;
56:     kprintf ("OK\r\n");
57:   }
58:
59:   void reset_coprocessor(void)
60:   {
61:     kprintf ("Resetting coprocessor...");
62:     outb(0xf0,0x0);
63:     __SLOW_DOWN_IO;
64:     outb(0xf1,0x0);
65:     __SLOW_DOWN_IO;
66:
67:     asm ("finit
68:     clts
69:     mov %cr0,%eax
70:     and $0xfffffff1,%eax
71:     mov %eax,%cr0
72:     ");
73:
74:     kprintf ("OK\r\n");
75:   }
76:
77:   void nibble_print(uint32 x)
78:   {
79:     kprintf("%c%c%c%c",
80:         x & 0xff , (x >> 8) & 0xff, (x >> 16) & 0xff, (x >> 24) & 0xff);
```

```
81:  }
82:
83:  void cpuid(void)
84:  {
85:    uint32  eax,ebx,ecx,edx;
86:    asm ("movl $0x0,%%eax;"
87:        "cpuid;"
88:        "movl %%eax,%0;"
89:        "movl %%ebx,%1;"
90:        "movl %%ecx,%2;"
91:        "movl %%edx,%3" : "=m" (eax) , "=m"  (ebx) , "=m" (ecx) , "=m" (edx));
92:    kprintf("CPU Type is `");
93:    nibble_print(ebx);
94:    nibble_print(edx);
95:    nibble_print(ecx);
96:    asm ("movl $0x1,%%eax;"
97:        "cpuid;"
98:        "movl %%eax,%0;"
99:        "movl %%edx,%1" : "=m" (eax) , "=m" (edx));
100:    kprintf("' Type %d, Family %d, Model %d, Stepping ID %d\r\n",
101:              (eax » 12) & 3, (eax » 8) & 0xf, (eax » 4) & 0xf, eax & 0xf);
102: }
103:
104: void banner(void)
105: {
106:   kprintf("                _       _      _       _ \r\n");
107:   kprintf("              | |     | |    | |     | |   \r\n");
108:   kprintf("              | |     | |    | |     | |    _\r\n");
109:   kprintf("              | |     | |     \\ \\  / /    |_|\r\n");
110:   kprintf("   _______    | |     | |      \\ \\/ /      _\r\n");
111:   kprintf("  /  ___  |   | |     | |      / /\\ \\     | |\r\n");
112:   kprintf("  | |   | |   | |     | |     / /  \\ \\    | |\r\n");
113:   kprintf("  | |   | |   | |____| |    | |    | |    | |\r\n");
114:   kprintf("  |_|   |_|   \\________|    |_|    |_|    |_|    "
115:                                       KERNEL_RELEASE " is started\r\n\n");
116: }
117:
118: void main_kernel_task(void * x)
119: {
120:   pthread_t t1;
121:   int i = 0;
122:
123:   kprintf("Main Task Started OK\r\n");
124:   driver_init();
125:   init_timer();
126:   cpuid();
127:   banner();
128:   kprintf("Starting User Tasks...\r\n");
129:   while (user_tasks[i].proc != NULL) {
```

```
130:      pthread_create(&t1,NULL,user_tasks[i].proc,NULL);
131:      i++;
132:    }
133:    current_task->status = TASK_SLEEPING;
134:    for (;;);
135:  }
136:
137:  void kernel_main()
138:  {
139:    console_base_init();
140:    kprintf ("\r\nNUXI release " KERNEL_RELEASE " loaded.\r\n");
141:    kprintf ("Copyright (C) 2001,2002 Corrado Santoro (csantodiit.unict.it)"
\
142:        " GPL Released.\r\n");
143:    kprintf ("Compiled on " __DATE__ "\r\n");
144:    kprintf ("Booting the kernel.\r\n");
145:    a20_enable();
146:    memory_test();
147:    reset_coprocessor();
148:    interrupt_setup();
149:    kprintf("Preparing Memory Manager...");
150:    // hummm.. we consider heap memory starting from second megabyte
151:    // indeed we should start from the end of kernel (how to compute it?)
152:    mm_init(0x100000,(uint32)machine_memory_size);
153:    kprintf("OK\r\n");
154:    //mailbox_init();
155:    kprintf("Preparing Task Structures...\r\n");
156:    task_add(NULL,main_kernel_task,NULL);
157:    // never return
158:    __cli();
159:    kprintf("PANIC! Unreachable point reached!\r\n");
160:    __halt();
161:  }
```

Chapter 6

The Console Display Manager

[TBD]

Chapter 7

Handling Interrupts

7.1 The Interrupt Descriptor Table

[Described in “x86 Basics” Chapter???]

7.2 Handling Hardware Interrupts

As we know from Intel Datasheets, processors of x86 family have three interrupt input pins: RESET, NMI and INT. The INT line is used for peripheral interrupt requests, and its management is performed by the processor using several steps, depicted in Figure 7.1. In particolar, when an interrupt signal (i.e. a transition from logic level 1 to logic level 0)[1] appears on the INT pin, the processor interrupts its activity, reads a value from its address bus, adds the IDT base and performs a long jump using the pointed gate (please note that each entry of the IDT is a gate)[2].

This technique is needed since processor has only one INT line but the request may arrive from different peripherals. Indeed, as Figure 7.2-left side shows, supposing that all peripheral interrupts lines are AND'ed together, each peripheral, after generating the interrupt signal, should present the proper address in order to allow processor to jump to the peripheral interrupt handling routine. Unfortunally, an AND gate is not sufficient, since it is not able to handle the simultaneous occurring of multiple interrupts, where a priority assignment is needed. To this aim, Intel designed an interrupt multiplexer chip, the 8259A called the Programmable Interrupt Controller, PIC in short. As Figure 7.2-right side shows, the PIC has 8 interrupt request input lines, numbered IRQ0-7, and it is connected to the processor though the bus and the INT signal. The PIC has two registers allowing programming: priorities can be assigned to IRQ lines as well

[1] The interrupt singnal is indeed a “pulse”: after tranistion from 1 to 0 the program is interrupted and the processor is “frozen”; when the signal returns to logic level 1, the interrupt service routine is started.

[2] This is the functioning in protected mode. In real mode, processor handles interrupts a little bit differently. However, NUXI works in protected mode, thus if you want to know how interrupts are managed in real mode, please refer to Intel Datasheets and Manuals.

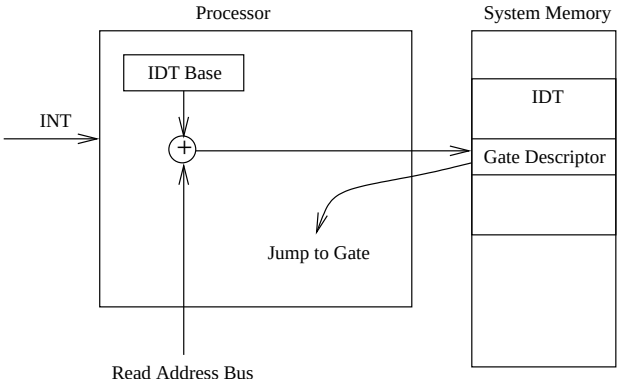

Figure 7.1: Interrupt Handling in Intel x86 Processors

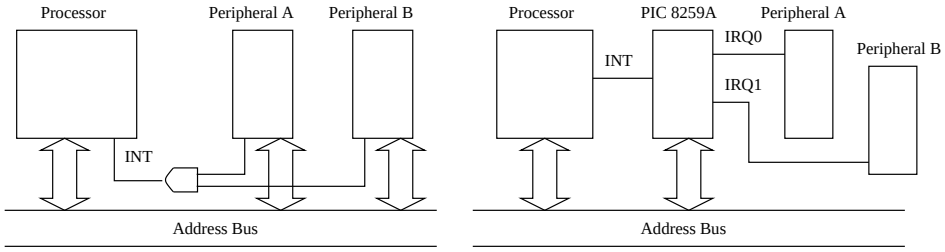

Figure 7.2: Multiple Interrupt Management in Intel x86 processors and 8259A

as the vector number to present on address bus for each IRQ signal. Thus each time an interrupt signal appears on an IRQ line, it is routeed to the INT processor pin by the PIC which puts also the programmed vector number on the address bus. Simultaneous IRQ requests are handled by the PIC by serving the one which has the highest priority, leaving the other pending until the former is completely served. For this reason, interrupt service routines must acknowledge interrupt requests to the PIC by writing a proper value on one of its registers (see below). This is generally done immediatelly before performing IRET istruction.

When more than 8 IRQs are needed, PICs can be connected in a master-slave fashion, as it happens in PCs since they have 16 interrupt lines on the bus. As reported in Figure 7.3, master-slave connections is performed by wiring the INT output of the master to a IRQ line of the slave (IRQ2 in the PC) and by programming PICs in order to enable master-slave functionality. Some of the depicted IRQ lines are then connected, in PCs, to pre-defined peripherals; this assingment is given in the right side of Figure 7.3.

7.3 Preparing Interrupt Management

Now it's time to know how to program our PICs. As we said above, a PIC has two 8-bit registers which are mapped to ports 020H - 021H, for the master PIC, and ports 0A0H - 0A1H, for the slave

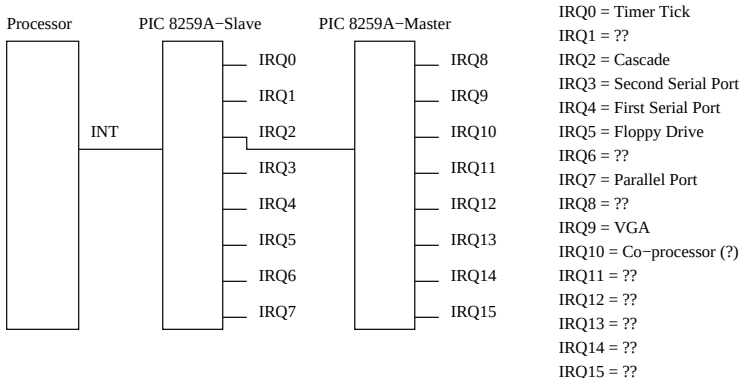

Figure 7.3: PICs and IRQ assignment in the PC

PIC. Initialization is performed by the `interrupt_setup` function in the source file "interrupt.c" and reported in Figure 7.4 (see Section 7.5 for a more detailed description of PIC initialization sequence). In particular, the first PIC (relevant to IRQ0-7) is initialized as master and enabled to generate INTs with vectors from 020H to 027H (according to the generated IRQ). On the other hand, the second PIC (IRQ8-15) is initialized to generate INTs with vectors from 028H to 02FH. We chose to map IRQs starting from vector 020H since it is the least available vector number: indeed, vectors from 00H to 01FH are used to map processor exceptions.

After initializing PICs, we mask off all IRQs (except IRQ2 which is used as the cascade line) since, till now, no interrupt service routine (ISR) is installed thus a interrupt which arrives at this stage could hang the system. IRQ masking is done by sutably setting register 1 of both PICs, it is a bit-mapped register where each bit controls the enabling of each IRQ line: if the bit is 1, the relevant IRQ line is disabled, otherwise it is enabled. In particular, register at port 021H controls lines from IRQ0 to IRQ7 (bit 0 = IRQ0, bit 1 = IRQ1, etc.), while register at port 0A1H controls the remaining lines (bit 0 = IRQ8, bit 1 = IRQ9, etc.).

After IRQ masking, we initilize the vector which contains the user-defined interrupt service routine to NULL (no ISR installed for each interrupt, see Sect. 7.4 below) and then we fill the interrupt descriptor table which we place at the beginning of memory (linear address 0)[3]. The `setup_idt_entry_int_gate` function creates an "interrupt gate" descriptor in the IDT: the first parameter is the vector number and the second and third parameters are the address of the service routine (offset and selector). We wrote 64 interrupt handlers, called from `_int00_handler` to `_int3f_handler` and defined in the assembler file "int.s", for interrupts from 00H to 03FH; these are used as "trampoline code" to call run-time defined interrupt service routines as we will see in the next section.

Finally, we load the IDT register by calling function `_load_idtr` (defined in "int.s"), enable interrupts (`__sli` is a macro defined in "include/asm.h") and exit.

[3]You can see, in the code of function `setup_idt_entry_int_gate`, that the `idt` variable which represent the pointer to IDT is always initialized to 0.

```
/*
 * interrupt.c
 * Copyright (C) 2001, 2002 Corrado Santoro (csanto@diit.unict.it)
 */

#define IRQ_BASE  0x20           // the base handler for the first irq
#define INTERRUPT_VECTORS  0x40  // number of entries in the IDT

void interrupt_setup(void)
{
  kprintf("Initializing interrupts units...");
  // initialize 8259A-1
  outb(0x11,0x20); __SLOW_DOWN_IO;      // initialization sequence
  outb(IRQ_BASE,0x21); __SLOW_DOWN_IO;  // irq 0-7 located at 0x20-0x27
  outb(0x04,0x21); __SLOW_DOWN_IO;      // 8259-1 is master
  outb(0x01,0x21); __SLOW_DOWN_IO;      // 8086 mode

  // initialize 8259A-2
  outb(0x11,0xa0); __SLOW_DOWN_IO;        // initialization sequence
  outb(IRQ_BASE+8,0xa1); __SLOW_DOWN_IO;  // irq 8-15 located at 0x28-0x2f
  outb(0x02,0xa1); __SLOW_DOWN_IO;        // 8259-2 is slave
  outb(0x01,0xa1); __SLOW_DOWN_IO;        // 8086 mode

  outb(0xff,0xa1); __SLOW_DOWN_IO;        // mask off all interrupts
  outb(0xfb,0x21); __SLOW_DOWN_IO;        // mask off interrupts but irq2 (cascade)
  // now setup the internal interrupt jump table
  for (i = 0;i < INTERRUPT_VECTORS;i++)
    int_services[i] = NULL;  // no ISR for each interrupt
// setup the handlers for each interrupt
  setup_idt_entry_int_gate(0x00,&_int00_handler,KERNEL_CODE_SELECTOR);
  setup_idt_entry_int_gate(0x01,&_int01_handler,KERNEL_CODE_SELECTOR);
//.....
  setup_idt_entry_int_gate(0x3e,&_int3e_handler,KERNEL_CODE_SELECTOR);
  setup_idt_entry_int_gate(0x3f,&_int3f_handler,KERNEL_CODE_SELECTOR);
  _load_idtr();
  __sti();
  interrupts_enabled = 1;
  kprintf("OK\r\n");
}
```

Figure 7.4: Preparing Interrupt Management in NUXI

7.4 The NUXI Interrupt Manager

As you can see in the source file "int.s", each `_intXX_handler` routine calls the `interrupt_handler` function (file "interrupt.c", Figure 7.5) passing as parameter the interrupt number. This function tests whether an ISR is defined in the `int_services` array for the requested interrupt number: if yes (the array item is not null), the ISR is called otherwise the message "unhandled interrupt 0xnn" is printed onto the screen and the system is halted. Thus adding an ISR to NUXI is simply made by setting the proper item of `int_services` array to the pointer of the ISR. This is done by the utility function `register_interrupt`, which takes, as parameter, the interrupt number to map and the pointer to the new interrupt service routine. The function also checks if an ISR is already defined for that interrupt. Registering ISR for IRQs is instead performed by the `register_irq` function. It behaves like `register_interrupt` but also provides IRQ line enabling by masking off the relevant bit the PIC. [PIC re-arming???]

7.5 8259A Basics

[Include this section???]

```
/*
 * interrupt.c
 * Copyright (C) 2001, 2002 Corrado Santoro (csanto@diit.unict.it)
 */

// the base handler for the first irq
#define IRQ_BASE  0x20
#define INTERRUPT_VECTORS 0x40

int_service_t  int_services[INTERRUPT_VECTORS];

// handles a generic interrupt
// It is called by the _intxx_handler (see int.s) defined in the IDT
void interrupt_handler(uint32 int_no)
{
  /* If a ISR is defined for this service, call it */
  if (int_services[int_no] != NULL)
    int_services[int_no]();
  else {
    kprintf("Unhandled interrupt 0x%x", int_no);
    __cli(); for (;;) ;
  }
}

// register a new ISR
int register_interrupt(int int_no,int_service_t routine)
{
  __cli();  // disable interrupts
  if (int_services[int_no] == NULL) {    // check whether the irq is already registered
    int_services[int_no] = routine;      // register the ISR
    __sti();                             // re-enable interrupts
    return 1;                            // return successfully
  }
  else {
    __sti(); return 0;
  }
}

// Register a new ISR relevant to an IRQ. Enables also the relevant bit of the 8259
int register_irq(int irq,int_service_t routine)
{
  __cli();  // disable interrupts
  if (int_services[irq+IRQ_BASE] == NULL) {   // check whether the irq is already registered
    int_services[irq+IRQ_BASE] = routine;     // register the ISR
    // now enable interrupt mask register of 8259
    if (irq < 8) {
      unsigned char mask = ~(1 << irq);      // reset the relevant irq bit
      outb(inb(0x21) & mask,0x21);
      outb(0x20,0x20);                       // re-arm PIC1
    }
    else {
      unsigned char mask = ~(1 << (irq-8)); // reset the relevant irq bit
      outb(inb(0xa1) & mask,0xa1);
      outb(0x20,0xa0);                      // re-arm PIC2
    }
    __sti();                                // re-enable interrupts
    return 1;                               // return successfully
  }
  else {
    __sti(); return 0;
  }
}
```

Figure 7.5: Interrupt Management in NUXI

Chapter 8

NUXI and Time

Managing flow of time is a fundamental part of any operating system, especially for a real-time one. Timers are used to trigger scheduling, to perform delays, to trigger execution of periodic tasks, to check deadline missing conditions. Timers are handled using a cooperation between software and hardware. A square-wave generator, calibrated to a suitable frequency, is connected to an IRQ thus generating interrupts at the given frequency. This interrupt is then used by the kernel (and in particular by the scheduling routine) to perform task switching and to maintain internal counters needed to emulate flow of time and to manage software-requested delays. In addition, in order to allow the kernel for autonomously calibrating the interrupt frequency, a special integrated circuit, called the *programmable timer/counter* (Figure 8.1, left-side), is connected between the square-wave generator and the IRQ pin; this circuit performs a division of the generator frequency by the value programmed, in its internal registers, by the software, in order to generate the timer/scheduling tick at the desidered frequency.

In PC-based platform, this integrated circuit is the Intel 8253 [rif], which is also called the *PIT - Programmable Interval Timer*. It is connected to a square-wave generator at the frequency of 1.193182 MHz (why this strange frequency value? It comes from the NTSC standard; probably the first IBM PCs could be connected to american television sets). The PIT embeds three counters which are used in PC platforms in the following manner (Figure 8.1, right-side): the first, counter 0, is used to generate IRQ0 for task scheduling and software timer management; the second, counter 1, is used to generate the refresh tick for the dynamic RAM; the third, counter 2, is instead connected to the system speaker and is used to generate the system beep. Generally, counter 1 and 2 are initialized by the BIOS and their divisor value is not modified by the operating system loaded. Counter 0 is instead re-initialized by the operating system according to the design choices. The PIT performs frequency division in the following way: initially, a counter is programmed by the OS according to its requirements (the divisor value), the counter is thus decremented at the rate of input frequency (1.193182 MHz), when it reaches zero the IRQ is generated and the counter is reloaded with the intially programmed value.

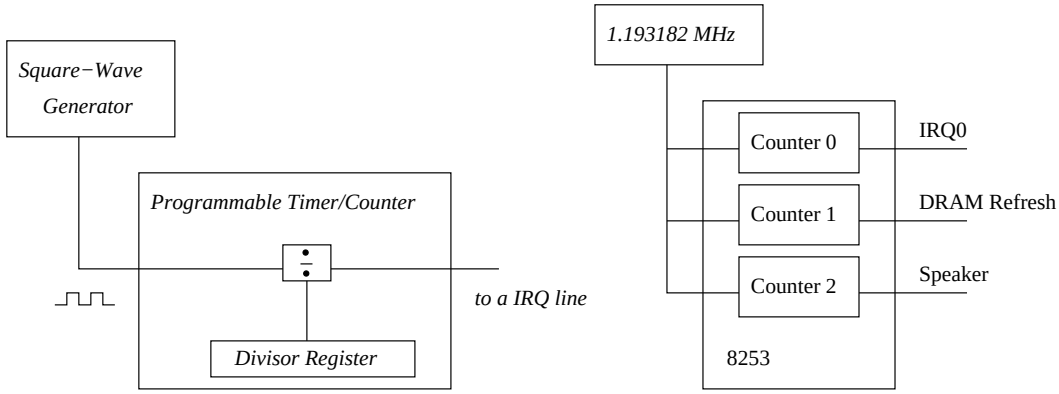

Figure 8.1: Programmable Timer/Counter

8.1 The 8253 System Timer

Now, before starting analyzing the timer modules of NUXI, let us spent some words on how the 8253 PIT can be programmed. Please note that here we describe only some functionalities of 8253, and, if you need further information, you may consult the relevant Intel documentation [rif.]

The 8253 of a PC-based platform is located at ports 040H-043H, with the meaning reported in Figure 8.2. The first three ports refer respectively to counter 0, 1 and 2, and are used to write the relevant divisor or to read the counter. The fourth port is the *control port*; it is bit-mapped, and the meaning of each bit is reported in Figure 8.2. As the figure shows, each timer can be programmed to work in various mode, but the one we are interested in is mode 3, which allows the generation of a square-wave with a frequency equal to the input frequency quotient the value programmed in the divisor. This value is loaded by issuing the command "read/write counter bits 0-7 and then 8-15" and then writing to the data port of the chosen counter first the LSB ♢ and then the MSB ♢ of the divisor. This operation, as regards counter 0, is performed in NUXI by the function `set_timer_resolution` in the source file "i8253.c" (lines 28–37, Section 8.5): it first outs the binary pattern `00110110` to the control port 043H (counter 0 select, read/write counter bits 0-7 and then 8-15, mode 3-square wave generator, binary 16-bit counter), then the LSB and finally the MSB of the resolution are set for counter 0, port 040H. The function `set_timer_resolution()` also calculates the value for the `USEC` variable which contains the timer period in microseconds; its value is then used to update the software timer and thus to manage flow of time.

Port	*Meaning*
040H	Counter 0 (IRQ0)
041H	Counter 1 (DRAM Refresh)
042H	Counter 2 (Speaker)
043H	Control Port

Control Port

Bits	*Meaning*
7-6	00 Counter 0 select 01 Counter 1 select 10 Counter 2 select
5-4	00 Count latch command 01 read/write counter bits 0-7 10 read/write counter bits 8-15 11 read/write counter bits 0-7 and then 8-15
3-1	000 mode 0 – interrupt on terminal count 001 mode 1 – programmable one shot x10 mode 2 – rate generator x11 mode 3 – square-wave generator 100 mode 4 – software triggered strobe 101 mode 5 – hardware triggered strobe
0	0 binary counter 16 bits 1 BCD counter

Figure 8.2: Registers of 8253 in PC-based platforms

8.2 8253 Initialization in NUXI

Timers initialization is performed, during NUXI startup phase, by the `init_timer()` function (lines 105–128, Section 8.6), which, in turn, calls `i8253_init()` (lines 49–56, Section 8.5) and then the function `set_timer_resolution()` to initialize the PIT. Function `i8253_init()` prepares the divisor value of counter 0 of the PIT by dividing the clock base frequency `CLOCK_BASE` by the `HZ` parameter, which is the scheduling tick frequency desidered. The values of `CLOCK_BASE` is defined in the include file "hz.h" while `HZ` variable is set in file "timer.c" (line 19). The default value is 100 Hz, which generates a clock interrupt each 10 milliseconds.

Function `init_timer()` performs also all the other operation needed to initialize time management: first of all it sets to zero the current time (line 109, see below); then it modify the entry 0x20 of the interrupt descriptor table (which corresponds to IRQ0 interrupt handler, see Chapter 7) by setting it as a *task state segment* in order to allow context saving of the interrupted task (lines 110–123, this will be more clearly explained in Chapter 9); then it registers the interrupt-service routine for IRQ0 (lines 125) and finally initializes the PIT (line 126).

The variable `curr_time` holds the current time and, in particular, the number of microseconds elpsed from last boot[1]; according to POSIX, it is of the type `struct timeval` (defined in "user/sys/time.h"), with the field `tv_sec` which stores the "seconds" part and the field `tv_usec` storing the "microseconds" part.

[1]Indeed this variable should hold the current (real) time, thus it should be initialized by reading the real-time clock at system boot. But this will be done in future releases of NUXI.

8.3 The Timer Handler Routine

The NUXI timer handler routine (i.e. the IRQ0 service routine), which is the function `tick_timer()` in "timer.c" (lines 33–37 of the source in Section 8.6), performs two main tasks: (1) it triggers the scheduler which handles task switching (function `scheduler()` of "task.c"), and (2) updates the current timer `curr_time` (function `do_timer()` of "timer.c", lines 39–79). If you take a look to the source code, you will find in `tick_timer()` only a call to `scheduler()`, because timer updating (i.e. call to `do_timer()`) is performed by a call issued inside the `scheduler()` function itself.

Current time updating performed by `do_timer()` must take into account that the scheduler is called not only when the timer tick occurs, but also when a thread goes to sleep awaiting for a resource (e.g. a semaphore, a condition, etc.). In this last case, as we will see in Chapter 10, the thread's state is set to `TASK_SLEEPING` by the wait function, and the scheduler is invoked immediatelly, in order to select a new ready thread. Therefore, when the scheduler is invoked due to a clock tick, we should increase the current time by the number of milliseconds stored in `USEC`; in the opposite case, since the sleeping thread does not have totally used its time quantum, we should increase the current time by the numer of milliseconds elapsed since the last clock tick. However, as Figure 8.3 shows, the situation could become more complex: indeed we must increase our current time by the microseconds elapsed *since last call to* `do_timer()`. To this aim, we need (1) to recognize whether the function is called due to a IRQ0 and (2) store the time elapsed since last call in a suitable variable. The first problem is solved by reading and testing the ISR register of the master PIC (lines 44–47). This register holds the number of IRQ which is currently begin served [cosa contiene se non ci sono IRQ pendenti??]: if its value is 0 the call is generated due to a IRQ0. The second problem is solved by using the variable `last_elapsed` which holds the time passed since last IRQ0 which is read from the PIT: to determine the time since last IRQ0 generation we read the current value of counter 0 (line 59), complement it to the divisor value (line 61, remember that the counter is decremented at the input frequency rate), and finally multiply it by the input frequency period (lines 62–70, `1000000/CLOCK_BASE` is the input frequency period in microseconds).

At this point, let us show how the mechanism to correctly increment current time works. Variable `last_elapsed` is first intialized to 0. If no task sleeping occurs and `do_timer()` is always called due to a IRQ0, the increment value is always `USEC` (lines 52–53). Now let us suppose that a task sleeping occurs: we thus calculate the time since last IRQ0 by reading the value of counter 0 of the PIT; then we assign this value to `last_elapsed` (lines 69–70, `last_elapsed` currenlty is 0) and use the same value to increment current time. Now two cases are possible: either next call to `do_timer()` is issued again by a thread going to sleep, or next call is triggered by IRQ0. In the latter case, the elapsed time is simply computed by subtracting `USEC` from `last_elapsed` (case 1 of Figure 8.3) and clearing `last_elapsed` since now we are synchronized again with IRQ0. In the former case, (case 2 of Figure 8.3) current time is update using the counter 0 value

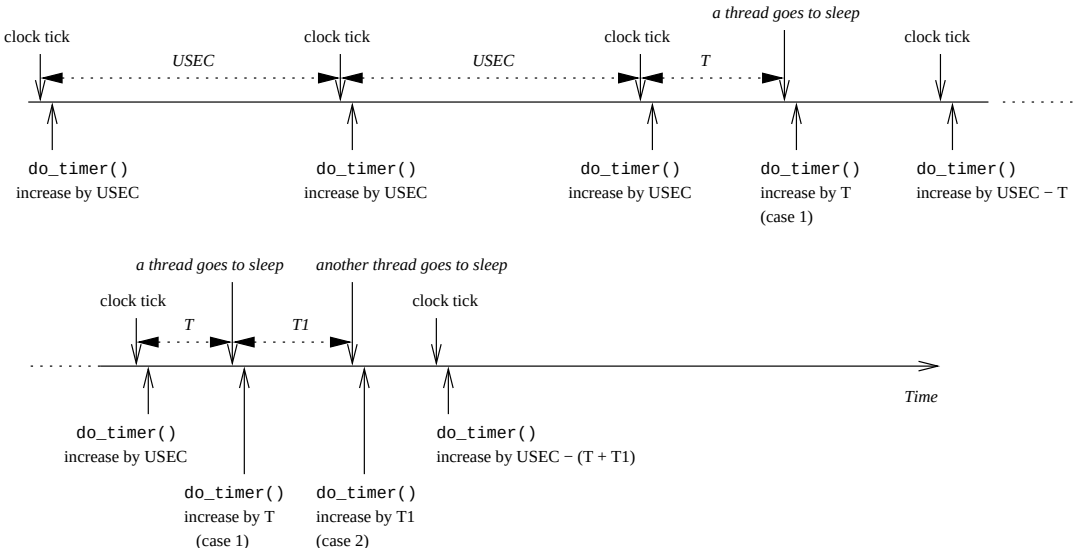

Figure 8.3: Update of current time in NUXI

(computed in microseconds in variable `val`) minus `last_elapsed` and setting the latter variable to `val` in order to reflect always the time passed since last IRQ0. Using this mechanism, correct evolution of current time is ensured.

8.4 NUXI Software Timers

[TBD]

8.5 8253 Management Source Code

```
 1: /*
 2:  * i8253.c
 3:  * Copyright (c) Dario Russo (dariorusso@hotmail.com)
 4:  */
 5: #include <kernel/kprintf.h>
 6: #include <kernel/asm.h>
 7: #define REALLY_SLOW_DOWN_IO
 8: #include <kernel/io.h>
 9: #include <kernel/interrupt.h>
10: #include <kernel/kernel.h>
11: #include <kernel/i8253.h>
12: #include <kernel/hz.h>
13:
14: // read the count register of the selected counter
15: unsigned int i8253_read(int counter)
```

```
16: {
17:   unsigned int lo,hi;
18:   counter &= 3;
19:   counter <<= 6;
20:  outb (counter,0x43); // 0x0 latch counter
21:  SLOW_DOWN_IO;
22:  lo = inb (0x40+counter);
23:  SLOW_DOWN_IO;
24:  hi = inb (0x40+counter);
25:   return hi * 256 + lo;
26: }
27:
28: inline void set_timer_resolution(volatile int resolution)
29: {
30:   USEC = 1000000.0*(float)resolution/(float)CLOCK_BASE;
31:  outb (0x36,0x43); //00110110b = 0x36
32:  SLOW_DOWN_IO;
33:  outb ((resolution & 0xff),0x40);
34:  SLOW_DOWN_IO;
35:  outb ((resolution >>8),0x40);
36:  SLOW_DOWN_IO;
37: }
38:
39: inline void set_timer_one_shot(volatile int countdown)
40: {
41:  outb (48,0x43); //00110000b = 48
42:  SLOW_DOWN_IO;
43:  outb ((countdown & 0xff),0x40);
44:  SLOW_DOWN_IO;
45:  outb ((countdown >>8),0x40);
46:  SLOW_DOWN_IO;
47: }
48:
49: inline void i8253_init(void)
50: {
51:   register long mipscounter = 0;
52:  kprintf("\r\nCopyright (C) Dario Russo (dariorusso@hotmail.com)i8253 Driver...\r\n");
53:
54:  set_timer_resolution(CLOCK_BASE/HZ);
55:  kprintf("Timers Initialized...");
56: }
57:
```

8.6 Timer Management Source Code

```
1: /*
```

```
 2:  * timer.c
 3:  * Copyright (C) Corrado Santoro (csanto@diit.unict.it)
 4:  */
 5:
 6: #include <kernel/ntypes.h>
 7: #include <kernel/interrupt.h>
 8: #include <kernel/io.h>
 9: #include <kernel/kprintf.h>
10: #include <kernel/asm.h>
11: #include <kernel/timer.h>
12: #include <kernel/task.h>
13: #include <kernel/kernel.h>
14: #include <kernel/i8253.h>
15: #include <kernel/hz.h>
16: #include <user/sys/time.h>
17:
18: extern uint32 _int20_handler;
19: int HZ = 100;
20: unsigned long USEC;
21: struct timeval curr_time;
22: task_struct timer_task;
23: long last_elapsed = 0;
24:
25:
26: /*
27:  * OK this is the interrupt call sequence
28:  * task -->
29:  *    .... isr (start.s)
30:  *    .... handle_interrupt(uint32)
31:  *    .... tick_timer
32:  */
33: void tick_timer(void)
34: {
35:   scheduler();
36:   outb(0x20,0x20);
37: }
38:
39: // updates the software current time
40: long do_timer(void)
41: {
42:   uint8 isr;
43:   long  val,incrval;
44:   // read the ISR register of 8259-1
45:   outb(0xa,0x20);   // 00001010 pattern to read ISR register
46:   isr = inb(0x20);
47:   outb(0x8,0x20);   // 00001000 reset read condition
48:   // test if IRQ0 is being serviced
49:   if (isr == 0)
50:   {
```

```
51:     // ok! increment software timer
52:     incrval = USEC - last_elapsed;
53:     last_elapsed = 0;
54:   }
55:   else
56:   {
57:     // do_timer is called due to a process which goes to sleep
58:     // read current count
59:     val = i8253_read(0);
60:     // determine the number of counts from last IRQ
61:     val = CLOCK_BASE/HZ - val;
62:     // determine the microseconds elapsed
63:     if (val < 2000)
64:       val = (val * 10000001)/CLOCK_BASE;
65:     else
66:       // prevent long overflow
67:       // FIXME! Use longmax!
68:       val = ((val * 10001)/(CLOCK_BASE/10001))*10001;
69:     incrval = val - last_elapsed;
70:     last_elapsed = val;
71:   }
72:   curr_time.tv_usec += incrval;
73:   if (curr_time.tv_usec > 10000001)
74:   {
75:     curr_time.tv_usec = curr_time.tv_usec % 10000001;
76:     curr_time.tv_sec++;
77:   }
78:   return incrval;
79: }
80:
81: // perform the sum result = result + operand
82: void time_add(struct timeval *result,struct timeval *operand)
83: {
84:   result->tv_usec += operand->tv_usec;
85:   if (result->tv_usec > 10000001)
86:   {
87:     result->tv_usec = result->tv_usec % 10000001;
88:     result->tv_sec++;
89:   }
90:   result->tv_sec += operand->tv_sec;
91: }
92:
93: // perform the difference result = result - operand
94: void time_diff(struct timeval *result,struct timeval *operand)
95: {
96:   if (result->tv_usec < operand->tv_usec)
97:   {
98:     result->tv_usec = result->tv_usec + 10000001 - operand->tv_usec;
99:     result->tv_sec--;
```

```
100:   }
101:   result->tv_sec -= operand->tv_sec;
102: }
103:
104: // Initialize timer structures and 8253 timer/counter
105: void init_timer(void)
106: {
107:   __cli();
108:   kprintf("Initializing Timer and Scheduler...");
109:   curr_time.tv_sec = curr_time.tv_usec = 0;
110:   // Timer Initialization
111:   // To allow scheduler to perform task switching, we map the INT 0x20 handler
112:   // (IRQ0) as a task gate. In this way, the interrupt task state
113:   // is saved by the processor and can be restored easily.
114:
115:   // Thus first prepare the timer_task TSS using the _int20_handler as
116:   // routine entry point
117:   task_prepare(&timer_task,(startup_proc)&_int20_handler,NULL);
118:   // Then prepare in the GDT a TSS descriptor pointing to prepared TSS.
119:   // It is associated to selector KERNEL_TIMER_TSS
120:   gdt_make_tss(KERNEL_TIMER_TSS,(uint32)&timer_task.tss,sizeof(timer_task),0);
121:   // Now setup the IDT entry relevant to INT 0x20 as a task gate descriptor
122:   // pointing to the KERNEL_TIMER_TSS selector
123:   setup_idt_entry_task_gate(0x20,0,KERNEL_TIMER_TSS);
124:   // finally register in the interrupt manager our IRQ0 handler
125:   register_irq(0,tick_timer);
126:   i8253_init();
127:   kprintf("OK\r\n");
128: }
129:
```

Chapter 9

Running Tasks

As stated in Section 2.2, the multiprogramming model of NUXI is single-process with multiple threads which we will simply call **tasks**. In the current version of NUXI, the task model is quite simplified. First of all, even if there is a system call handler, there is no distinction between user level and kernel level and also memory space is not protected: both kernel and user programs run at CPU privilege 0 (the most privileged level), and see the same linear address space of 4 GBytes (see Section 5.2). This is basically due to the fact that user programs are linked together in a single large binary which is entirely loaded during boot phase. Since this mode is not a good functioning technique for OSs, we plan to modify it in future versions of NUXI. The absence of memory separation does not require to maintain some per-task information, such as memory region or memory page tables, but all the other data regarding task state need to be managed. To this aim, the *context of a task* is stored in special structures, of the type `task_struct` (lines 54–65, Section 9.7), composed of two parts: the CPU-dependent context – i.e. CPU registers –, the OS-dependent context – i.e. the stack space, the list of opened files, etc. All running tasks are thus represented by a set of `task_structs` linked together in two lists, ordered respectively by *priority* and by *task identifier*. The former parameter gives the policy of choicing the task to be run among various ready tasks; currenly NUXI manages up to 256 priority levels. This latter parameter, also called the *TID*, is a unique number which identifies the task; it has the same meaning of the PID of a Unix system.

All task management routines are present in the source file “task.c”, whose listing is given in Section 9.8, while Section 9.7 reports the relevant include file (task.h). These routines perform essentially task creation, task destruction, scheduling and dispatching. Since NUXI is designed to run onto PC-based platforms, we exploited the native task switching mechanism, offered by Intel processors, to perform CPU context saving and restoring. The functioning of this mechanism, together with the other task management operations, are described in the following Sections.

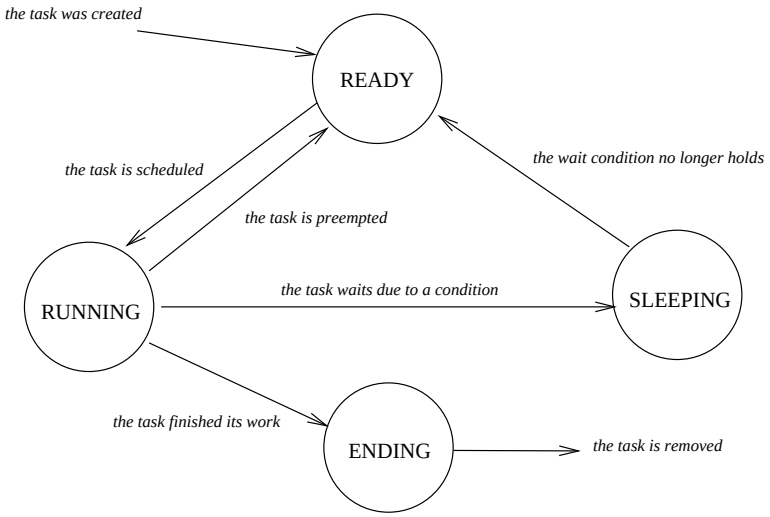

Figure 9.1: Task's State and Task Evolution

9.1 Task States

Figure 9.1 depicts the finite state machine which describes the evolution of the state of a task; it is quite similar to those of other existing OSs. A task state in NUXI can assume one of the following values, which are defined as constants in file "task.h" (lines 13–18, Section 9.7):

RUNNING (TASK_RUNNING) The task is currently running.

READY (TASK_READY) The task is ready to run, but it is not running.

SLEEPING (TASK_SLEEPING) The task is waiting for something (a delay, a semaphore, a condition variable, etc.).

ENDING (TASK_ENDING) The task finished its work (the task's main routine is ended).

State transitions are triggered by events shown in Figure 9.1. Basically, when a task is created, it is in the READY state and goes to RUNNING when it is selected by the scheduler. Transition from RUNNING to READY occurs when the scheduler performs task preemption in order to run another ready task. If the task has instead issued a command which provokes a suspension, such as a delay, waiting for a semphore, or a condition variable, its state goes to SLEEPING and then again to READY when the wait condition no longer holds. Finally, when a task ends, its state goes to ENDING; in this case, the task is physically removed and the relevant structures are released by the scheduler itself.

9.2 The Task Structure and Task Lists

As introduced above, in NUXI the context of a task is stored in a structure of the type `task_struct`, which is defined in the include file "kernel/task.h", the code of which is reported in Section 9.7 (lines 54–65). Each field of this structure has the following meaning:

tss. This is the so-called *task state segment*, and is the CPU-dependent part of the task context. It is defined in the lines 21–39, and reflects the same structure of the task state segment managed by Intel processors. Its precise structure and the functioning of the task switching mechanism is reported in Section 9.6.

stack. An array which contains the task's stack. In the current version of NUXI its size is fixed to 4096 (`TASK_STACK_SIZE`, line 42), which should be enough.

tid. The task identifier (TID), a unique number which identify this task.

p_tid. The TID of the parent task, i.e. the task which created this task.

state. The state of the task; it can be one of the constants defined in lines 13–18 and it was explained in the previous Section.

files. It is an array of `file *` and holds information on files opened by the task. Its use is explained in Chapter 12.

cpu_time. The amount of CPU time spent by the task in the running state.

syscall. A structure holding the information needed to handle system calls. Explained in ????.

errno. The "errno" variable for this task/thread. It holds the error number of the last failed system call.

next, next_ordered. Two pointers to another `task_struct` used to maintaint a linked list of tasks of the same priority, and a linked list ordered by TID. Explained below.

To keep track and manage all tasks in the system, NUXI maintains two internal structure which are represented in Figure 9.2: the `tasks` array (defined in line 27, Section 9.8) and the `task_list` (the type is defined in lines 67–71, Section 9.7, while the variable is defined in line 23, Section 9.8). The former is an array of `task_struct` pointers where each element represents a circular linked list (called the *priority list*) of all `task_structs` referring to tasks with the same priority: the element n is the list of tasks with priority n. All `task_structs` in a priority list are linked together through the `next` field. If there are no task for a certain priority, the relevant entry of the `tasks` array is set to `NULL`. Therefore, to know all the tasks which run at a given priority, we simply access the relevant element of the array and walk the priority list through the `next` field.

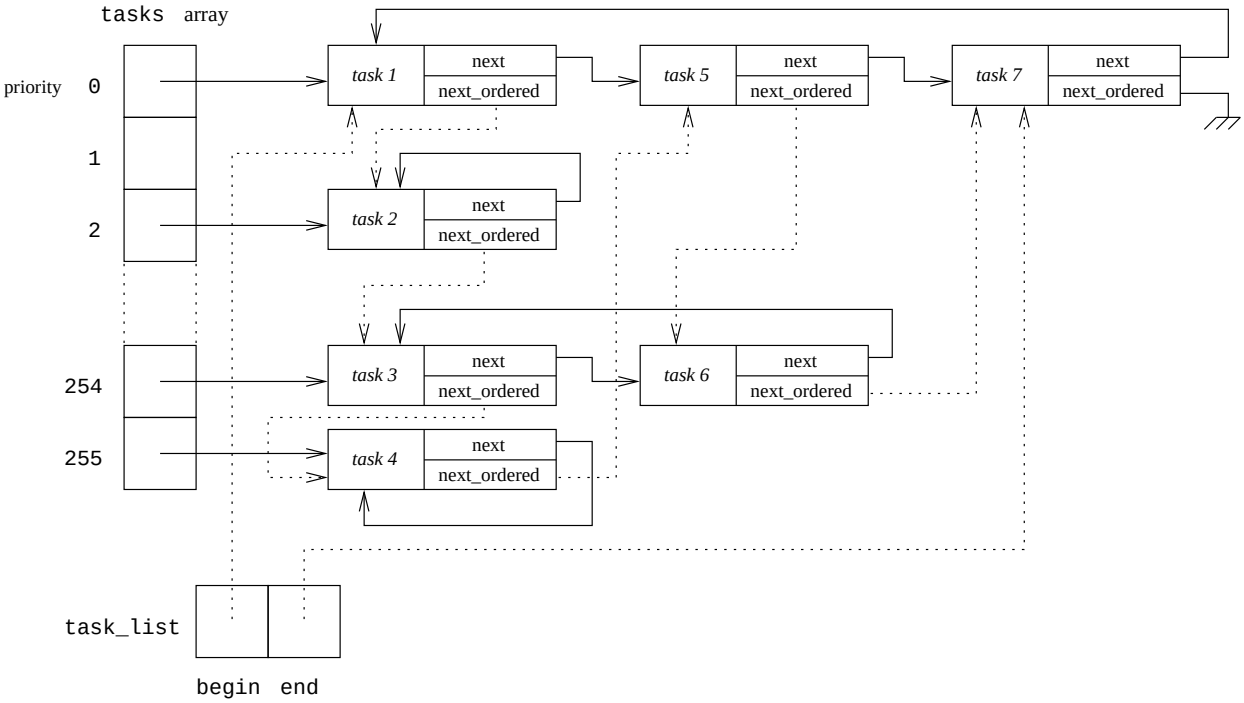

Figure 9.2: NUXI Task Lists

The second structure (`task_list`) represents a list of all tasks ordered by TID, which is linked through the `next_ordered` field. The `task_list` variable has two fields, `begin` and `end`, which point respectively to the first and the last element of the list. In order to obtain a list of tasks ordered by TID we access the `begin` field of `task_list`, and then walk the list through the `next_ordered` field until `NULL` is reached.

The practical utilization of these two structures is the following. By maintaining a priority order simplify the process of selecting the ready task with the highest priority, which is performed by the scheduler. The TID-ordered list, instead, allows to obtain the list of tasks ordered by *creation epoch*, since a new TID is generated when a new task is started up and it is never reused (unless there were be more than 4 billions task creations since power up, and thus TID number, which is 32-bit unsigned, restarts from zero).

9.3 Task Scheduling

As it is widely known, the scheduler is that part of an OS entailed to select the next task to run. In NUXI this is performed by the `scheduler()` function of “task.c” (lines 150–251, Section 9.8). Even if this function could seem long, it is quite simple and aims to scan the `tasks` array in order to:

1. select the task in READY state with the highest priority;
2. remove the entries relevant to tasks in ENDING state.

Once the task to be run is selected, the variable `current_task` (defined in line 26, Section 9.8) is set to the selected task's `task_struct`, in order to allow the kernel to refer to the currenlty running task simply by using this variable. Now let us analyze the scheduler code.

First of all, the `current_task` variable is tested against `NULL` (running the scheduler with no current task does not make sense). In this case a panic message is displayed and the system is halted. Secondly, we update the `cpu_time` of the running task (lines 162–164) by obtaining – by the function `do_timer()` – the number of microseconds elapsed since last scheduler call (see Chapter 8 for the details on time handling). Then, if the scheduler is invoked due to a task preemption we have to change the state of the current task from RUNNING to READY (line 165). The test is needed since the scheduler can be invoked not only by the clock interrupt handler but also by functions which cause a wait condition (e.g. downing a semaphore, see Chapter 10); in this last case, the scheduler is entered with the state of the current task already set to SLEEPING, thus a state change to READY must not occur. At this point, the code in lines 167–250 selects the READY task with the highest priority by scanning each priority list. Please note that (using the same convention adopted in UNIX), priority number 0 is the highest while 255 is the lowest, thus we have to scan priority lists starting from the first entry of the `tasks` array (loop at line 167). First the scheduler tests whether the selected entry of the array (the head of the priority list) contains a list (i.e. it is not `NULL`) and then scans this list ("do-while" loop at lines 174–245) in order to find the first READY task. Once a task is found (lines 178–192), before running it, the head the the scanned priority list (the entry of the `tasks` array) is set to the task to be run (line 181) in order to start the scan, at the next scheduler call, from the subsequent task. In this way, we implement a round-robin mechanism on each priority list. Then the task is run (lines 181–189) using the dispatching mechanism provided by the task management of Intel processors (see Section 9.6). Indeed, the task is run by the function `task_call()` called at line 189, which performs the task switch: the scheduler is suspended here, i.e. the next time it will be called (due to a IRQ0, for example), its execution will start from here, not from the beginning of the function. This is due to the task switching mechanism of the Intel process, which will be deailed in the following Sections of this Chapter. For this reason, we placed the "goto" statement at line 191 which forces a jump to begin of the scheduler routine.

During priority lists scanning, if the scheduler finds a task in the ENDING state, the latter is removed by adjusting the links in both the priority list and the task list, freeing also the allocated memory (lines 193–242).

If the scheduler did not find a READY task in the scanned priority list, the "do-while" loop at lines 174–245 ends, and the scheduler starts to scan the next priority list. If all the lists were scanned, without finding a READY task, the program will/would reach line 249, which prints the panic message and halts the system: indeed, in all OSs, when there are no ready tasks (i.e.

all tasks in SLEEPING state) the so-called “idle process” is run, but in NUXI the “idle process” is not invisible, like in other OSs, and it is a task (which does nothing) with its `task_struct` entry linked in the task management lists. The idle task is always in the RUNNING/READY state, thus, in our case, to have to no ready tasks does not make sense.

9.4 Task Switching in Intel x86 Family

Since the introduction of 80286 CPU, Intel processors provide a native mechanism for task switching in order to rely OS code to save the context of a pre-empted task and to restore the context of the task to be scheduled. This mechanism works only when the processor is in protected mode and is performed by using the so-called **Task State Segments** or **TSS**. A TSS is a memory region (Figure 9.3) which contains the complete CPU context in terms of register values, a user-defined area and other information regarding the priviledges for accessing I/O ports [ref. Intel manual]. This memory region becomes a TSS, for the CPU, when it is referred by a “Task State Segment” entry in the GDT (or LDT); here the base address is the beginning linear address of the TSS region in memory, the size is the region length in bytes, while the entry type is “32-bit TSS”. Any long jump or long call instruction with a destination selector which refers to a TSS causes a *task switch*, i.e. the CPU registers are saved in the current task’s TSS and the values stored in the called/destination TSS are loaded in the registers, thus continuing execution to the location referred by CS:EIP of the new TSS. The TSS of the currently running task is stored in a special register called TR (task register) which contains the selector (in the GDT or LDT) of the current TSS. Let us explain more deeply this process. Let us suppose that the fourth entry of the GDT (selector 018H) refers to a TSS and that the program makes a long jump or long call to 018H:0000H[1]. In this case the CPU peforms the following steps:

1. It checks that current TR refers to a TSS. If no, it means that this is the first task switch since power up, thus step 2 is skipped.
2. It saves the values of all registers in the TSS referred by TR.
3. It loads TR register (task register) with value 018H which points to the selector of new TSS.
4. It loads all its registers using the values stored in the TSS referred by TR (which now is 018H). Therefore execution will continue from location referred by the values of CS:EIP stored in the TSS.
5. Performs other operations in order to keep updated the task chain (see Intel manual [rif!]).

[1]Please note that when a destination location refers to a TSS, the offset part is ignored since the real destination address is taken from the CS:EIP registers stored in the TSS itself.

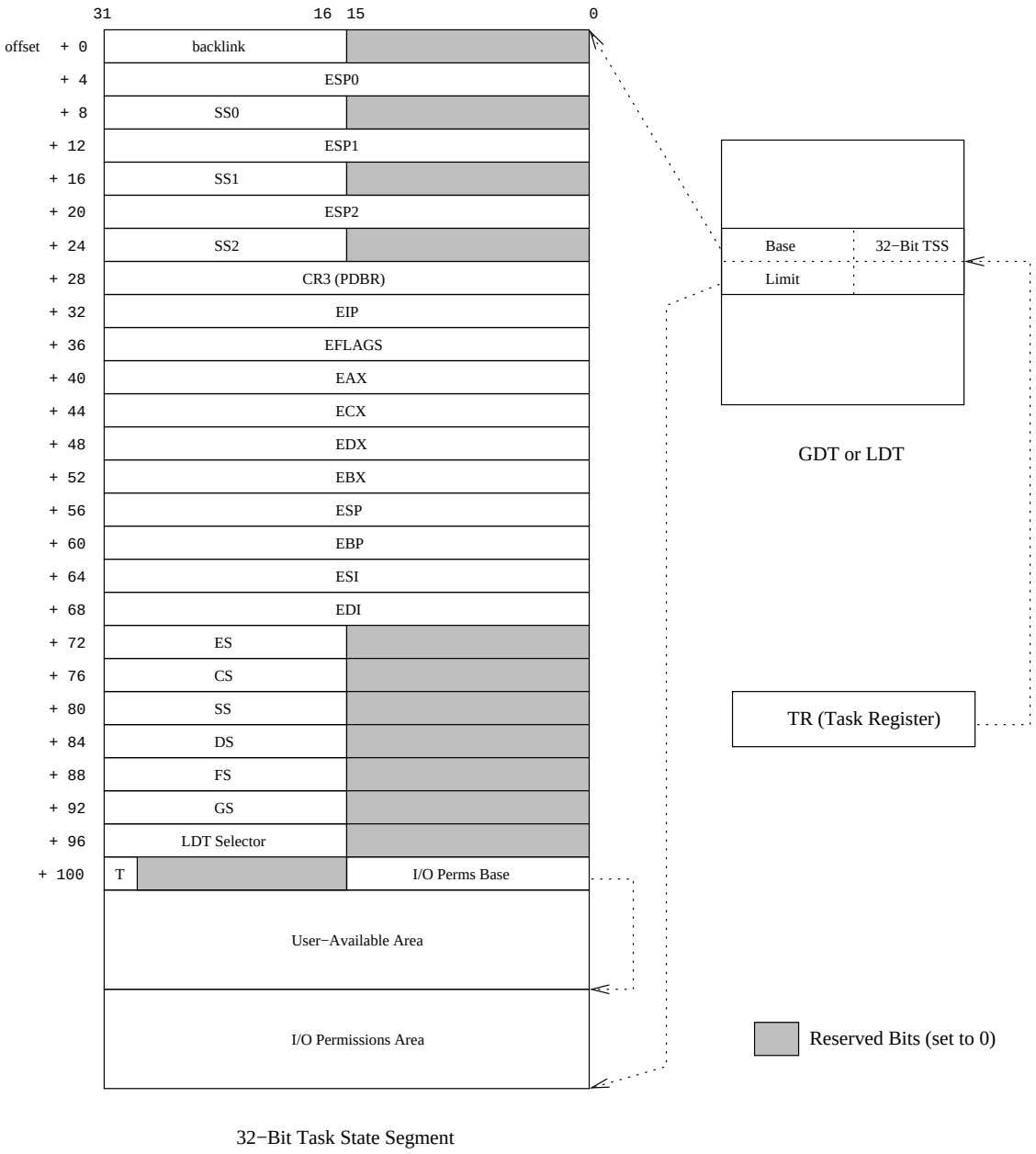

Figure 9.3: A Task State Segment

NUXI uses this CPU facility to perform task switching, which is triggered, in particular, by the timer tick (IRQ0). However, this makes NUXI code CPU-dependent requiring to make many patches in order to make it run on other platforms. But this is a design choice since NUXI is developed for x86 platforms. Other OSs, such as Linux which is cross-platform, implement task switching totally in software [rifs!].

9.5 Starting a New Task

9.6 Task Switching in NUXI

9.7 "task.h" Header File Source

```
 1: /*
 2:  * task.h
 3:  * Copyright (C) Corrado Santoro (csanto@diit.unict.it)
 4:  */
 5:
 6: #ifndef __TASK_H
 7: #define __TASK_H
 8:
 9: #include <kernel/ntypes.h>
10: #include <kernel/file.h>
11: #include <user/sys/time.h>
12:
13: #define TASK_STARTUP    0
14: #define TASK_RUNNING    1
15: #define TASK_READY      2
16: #define TASK_SLEEPING   3
17: #define TASK_PREEMPTED  4
18: #define TASK_ENDING     5
19:
20: #pragma pack(1)
21: typedef struct TSS32 {
22:    uint16 link, __unused0;
23:    uint32 esp0;
24:    uint16 ss0, __unused1;
25:    uint32 esp1;
26:    uint16 ss1, __unused2;
27:    uint32 esp2;
28:    uint16 ss2, __unused3;
29:    uint32 cr3, eip, eflags;
30:    uint32 eax,ecx,edx,ebx,esp,ebp,esi,edi;
31:    uint16 es, __unused4;
32:    uint16 cs, __unused5;
33:    uint16 ss, __unused6;
```

```
34:    uint16 ds, __unused7;
35:    uint16 fs, __unused8;
36:    uint16 gs, __unused9;
37:    uint16 ldt, __unused10;
38:    uint16 debugtrap, iomapbase;
39: } TSS;
40: #pragma pack()
41:
42: #define TASK_STACK_SIZE   4096
43:
44: typedef void (*startup_proc)(void *);
45:
46: #define MAX_SYSCALL_PARAMS  32
47:
48: typedef struct {
49:   uint32  syscall_no;
50:   uint32  params[MAX_SYSCALL_PARAMS];
51:   uint32  return_value;
52: } t_syscall_p;
53:
54: typedef struct task_struct {
55:   TSS     tss;
56:   uint8   stack[TASK_STACK_SIZE];
57:   uint32  tid;
58:   uint32  p_tid;
59:   uint8   state;
60:   file  * files[MAX_FILES];
61:   struct timeval  cpu_time;
62:   t_syscall_p  syscall;
63:   int     errno;
64:   struct task_struct * next,* next_ordered;
65: } task_struct;
66:
67: typedef struct {
68:   task_struct * begin;
69:   task_struct * end;
70:   int total_tasks,last_tid;
71: } t_task_list;
72:
73: extern task_struct * current_task;
74: extern t_task_list task_list;
75:
76: void task_init(void);
77: void task_prepare(task_struct * task,startup_proc proc,void * param);
78: void task_add(uint32 *tid,startup_proc proc,void * param);
79: void task_end(void);
80: void task_dump(void);
81: void scheduler(void);
82:
```

```
 83: #endif
```

9.8 Task Management Source Code

```
  1: /*
  2:  * task.c
  3:  * Copyright (C) 2001, 2002 Corrado Santoro (csanto@diit.unict.it)
  4:  */
  5:
  6: #include <kernel/ntypes.h>
  7: #include <kernel/io.h>
  8: #include <kernel/asm.h>
  9: #include <kernel/interrupt.h>
 10: #include <kernel/kprintf.h>
 11: #include <kernel/kernel.h>
 12: #include <kernel/task.h>
 13: #include <kernel/gdt.h>
 14: #include <kernel/mm.h>
 15: #include <kernel/timer.h>
 16: #include <kernel/profile.h>
 17: #include <kernel/hz.h>
 18:
 19: t_profile_info profile_info;
 20:
 21: int pri = 10;
 22:
 23: t_task_list task_list = { NULL,NULL,0,0};
 24:
 25: #define MAX_PRIORITY  256
 26: task_struct * current_task = NULL;
 27: task_struct * tasks[MAX_PRIORITY];
 28:
 29: void task_call(uint32 tss)
 30: {
 31:  uint32 sel[2];
 32:   sel[0] = 0;
 33:  sel[1] = tss;
 34:   // enable 8259-1 timer interrupt and jump to new task
 35:  asm("mov $0x20,%%ax;out %%al,$0x20;sti;ljmp %0;cli"::"m" (*sel));
 36: }
 37:
 38: inline void task_prepare(task_struct * task,startup_proc proc,void * param)
 39: {
 40:   register uint32 * sp;
 41:
 42:   kmemset((uint8 *)&task->tss,sizeof(TSS),0);
```

```
43:  sp = (uint32*)((uint32)&task->stack + TASK_STACK_SIZE - 8);
44:   *sp = (uint32)param;
45:   sp--;
46:  task->tss.esp0 = (int32)sp;
47:  task->tss.ss0 = KERNEL_STACK_SELECTOR;
48:  task->tss.eax = 0;
49:  task->tss.ebx = 0;
50:  task->tss.ecx = 0;
51:  task->tss.edx = 0;
52:  task->tss.esi = 0;
53:  task->tss.edi = 0;
54:  task->tss.esp1 = task->tss.esp2 = task->tss.ss1 = task->tss.ss2 = 0;
55:  task->tss.cr3 = 0;
56:  task->tss.eip = (uint32)proc;
57:  task->tss.eflags = 0x0202; // 0x4202 -> paging on
58:  task->tss.esp = task->tss.esp0;
59:  task->tss.ss = task->tss.ss0;
60:  task->tss.cs = KERNEL_CODE_SELECTOR;
61:  task->tss.ds = task->tss.es =
62:   task->tss.fs = task->tss.gs = KERNEL_DATA_SELECTOR;
63:  task->tss.ldt = task->tss.debugtrap = task->tss.iomapbase = 0;
64: }
65:
66: // Start a new task!
67: void task_add(uint32 * tid,startup_proc proc,void * param)
68: {
69:   task_struct * task,* saved_current;
70:   // clear interrupt to prevent task switching
71:   __cli();
72:   // first allocate memory for task struct
73:   task = (task_struct *)kmalloc(sizeof(task_struct));
74:   if (task == NULL)
75:   {
76:     kprintf("No more space for tasks\r\n");
77:     __sti();
78:     return ;
79:   }
80:   // OK, first prepare the TSS in the task struct, using "proc" as
81:   // entry point
82:   task_prepare(task,proc,param);
83:   // Now map the KERNEL_TSS selector as a Task State Segment descriptor
84:   gdt_make_tss(KERNEL_TSS,(uint32)&task->tss,sizeof(task_struct),0);
85:   //display_gdt(KERNEL_TSS);
86:   // link new task in the ordered task list
87:   task->next_ordered = NULL;
88:   if (task_list.begin == NULL)
89:   {
90:     int i;
91:     // this is the first task
```

```
 92:     for (i = 0;i < MAX_PRIORITY;i++) tasks[i] = NULL;
 93:     task_list.begin = task;
 94:     task_list.end = task;
 95:   }
 96:   else
 97:   {
 98:     task_list.end->next_ordered = task;
 99:     task_list.end = task;
100:   }
101:   // link new task in the priority queues
102:   if (tasks[pri] == NULL)
103:   {
104:     tasks[pri] = task;
105:     task->next = task;  // circular list, link with itself
106:   }
107:   else
108:   {
109:     task->next = tasks[pri]->next;
110:     tasks[pri]->next = task;
111:   }
112:   // increment the total numer
113:   ++task_list.total_tasks;
114:   ++task_list.last_tid;
115:   // set the tid and the parent tid
116:   task->tid = task_list.last_tid;
117:   if (current_task == NULL)
118:     task->p_tid = 0;
119:   else
120:     task->p_tid = current_task->tid;
121:   if (tid != NULL) *tid = task->p_tid;
122:   task->cpu_time.tv_sec = 0;
123:   task->cpu_time.tv_usec = 0;
124:   // Update current_task. The created task will be scheduler immediatelly
125:   saved_current = current_task;
126:   saved_current->state = TASK_READY;
127:   current_task = task;
128:   current_task->state = TASK_RUNNING;
129:   // Now run the new task by performing a long jump using
130:   // the selector KERNEL_TSS
131:   task_call(KERNEL_TSS);
132:   current_task = saved_current;
133:   __sti();
134: }
135:
136: // end current task
137: void task_end(void)
138: {
139:   __cli();
140:   current_task->state = TASK_ENDING;
```

```
141:   scheduler();
142: }
143:
144: void task_dump(void)
145: {
146:   kprintf(" %x ",current_task->tss.eip);
147: }
148:
149: // This is the scheduler! It is very simple!
150: void scheduler(void)
151: {
152:   task_struct * prev,* end,* p;
153:   struct timeval temp;
154:   int i;
155: redo:
156:   if (current_task == NULL)
157:   {
158:     kprintf("PANIC! Scheduler is trying to run without tasks!\r\n");
159:     __cli();
160:     for (;;) ;
161:   }
162:   temp.tv_usec = do_timer();
163:   temp.tv_sec = 0;
164:   time_add(&current_task->cpu_time,&temp);
165:   if (current_task->state == TASK_RUNNING) current_task->state = TASK_READY;
166:   // OK! This is a simple round-robin with priorities
167:   for (i = 0;i < MAX_PRIORITY;i++)
168:   {
169:     current_task = tasks[i];
170:     if (current_task != NULL)
171:     {
172:       end  = current_task;
173:       prev = NULL;
174:       do
175:       {
176:         prev = current_task;
177:         current_task = current_task->next;
178:         if (current_task->state == TASK_READY)
179:         {
180:           // We found the task to run
181:           tasks[i] = current_task;
182:           // Since we use always the same selector to run the current_task,
183:           // let's update the KERNEL_TSS selector in the GDT with the TSS
184:           // of the new task to be scheduled
185:           gdt_make_tss(KERNEL_TSS,
186:                        (uint32)&current_task->tss,sizeof(task_struct),0);
187:           current_task->state = TASK_RUNNING;
188:           // well, switch to the new task
189:           task_call(KERNEL_TSS);
```

```
190:           // then restart scheduler
191:           goto redo;
192:         }
193:         else if (current_task->state == TASK_ENDING)
194:         {
195:           // task is marked to be terminated, remove it
196:           // first remove it from the ordered list
197:           task_struct *pprev;
198:           p = task_list.begin;
199:           pprev = NULL;
200:           // walk the ordered list to find the TID
201:           while (p != NULL)
202:           {
203:             if (p->tid == current_task->tid)
204:             {
205:               // TID found
206:               if (p == task_list.begin)
207:               {
208:                 // the element is the first
209:                 task_list.begin = p->next_ordered;
210:               }
211:               else if (p == task_list.end)
212:               {
213:                 // the element is the last
214:                 task_list.end = pprev;
215:                 if (pprev != NULL) pprev->next_ordered = NULL;
216:               }
217:               else
218:               {
219:                 // the element is in the middle
220:                 pprev->next_ordered = p->next_ordered;
221:               }
222:               task_list.total_tasks--;
223:               break;  // break the while
224:             }
225:             pprev = p;
226:             p = p->next_ordered;
227:           }
228:           // now remove the task from the priority list
229:           p = current_task;
230:           if (current_task->next == current_task)
231:           {
232:             // no more task in this priority list
233:             tasks[i] = NULL;
234:             kfree(p);
235:             break; // break do-while
236:           }
237:           else
238:           {
```

```
239:             prev->next = current_task->next;
240:             current_task = current_task->next;
241:             kfree(p);
242:           }
243:         }
244:       }
245:       while (current_task != end);
246:       // we walked the entire priority list without finding a ready task
247:     }
248:   }
249:   kprintf("PANIC! Scheduler did not found a ready tasks. Cannot continue!\r\n");
250:   __halt();
251: }
```

Chapter 10

Handling Concurrency

In any multiprogrammed operating system, all problems related to concurrency and race conditions are faced with the use of appropriate structures which handle access to critical section, to manage sleep/wakeup conditions, etc. In NUXI these structures are *semaphores*, which are modeled according to the Dijkstra's generalized semaphores [rif!], and *condition variables*, which allow a form of task synchronization based on a condition which must hold in order to enter a critical section. Condition variables behave very similar to the structures used in the POSIX pthread package [rif!]. Both semaphores and condition variables are based on other internal structures, called *wait channels*, which handle the sleep/wakeup mechanism.

The source code of all the routines for concurrency handling is included in the file "wait.c" (reported in Section 10.5) and the relevant header file is "kernel/wait.h" (reported in Section 10.4).

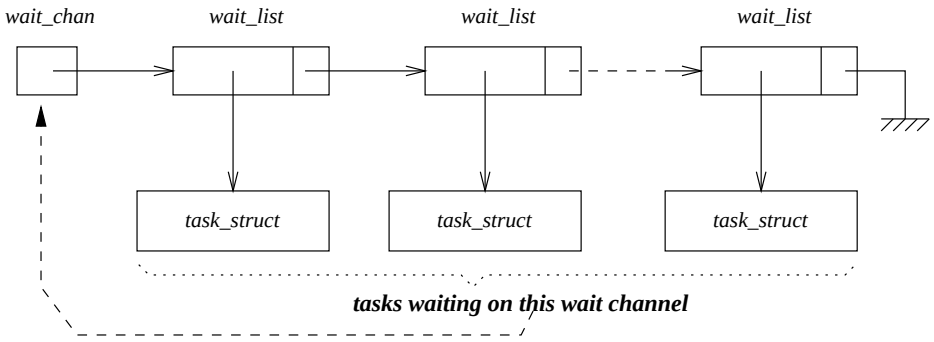

Figure 10.1: Wait Channels.

10.1 Wait Channels

A NUXI wait channel is a linked list of tasks waiting for the occurence of the same condition. Its type is `wait_chan` (defined in line 32, Section 10.4) which represents a pointer to the beginning of the list (Figure 10.1). Each element (`wait_list`, lines 27–30, Section 10.4) is structure composed of a pointer to the `task_struct` representing the task in the list, and a pointer to the next element of the list. Of course, the list is ended with a NULL.

A task can suspend itself by invoking the function `thread_sleep()` (lines 43–48, Section 10.5), passing as parameter a `wait_chan` pointer which represents the channel onto which to perform suspension. Then, when another task invokes the function `thread_wakeup()` (lines 62–67, Section 10.5), all the tasks suspended onto the channel, passed as parameter, are awaken, i.e. placed in the READY state in order to be selected by the scheduler. The source code of these functions is quite simple: they both simply perform a call respectively to the functions `in_int_sleep()` and `in_int_wakeup()` with interrupts disabled. As we will see in the folloiwng, the reason for these indirected calls relies on the fact that, for semaphores and condition variables management, we need to handle “sleep” and “wakeup” with interrupt disabled, thus we need suitable functions which do not change the interrupt enable processor flag. The `in_int_sleep()` routine (lines 27–41, Section 10.5) performs the following steps: it allocates the memory for a new element of the `wait_list` (lines 30–34), then it sets to SLEEPING (line 35) the state of the current task (i.e. the task invoking this sleeping function), then the created element is filled with the proper information (lines 36–37) and is placed at the beginning of the list represented by the `wchan` parameter (line 38); finally the scheduler is invoked (line 40) by a software interrupt call to vector 0x20, operation which really causes the task to be suspended. This software interrupt call is equivalent to a timer tick (IRQ0, mapped to vector 0x20, see Chapter 7), thus causing the pre-emption of the current task (saving of the task’s state) and a call to the scheduler (Section 9.6) which then selects another task to run. In this way, to wake up a sleeping task implies simply to change its state to READY: when the scheduler will select it, its execution will resume immediatelly after the INT 0x20 call (line 41, return to caller). This is done by the `in_int_wakeup` function (lines 50–60), which scans and clears the list pointed by the parameter `wchan`, changing to READY the state of each task stored, in order to wake up it.

10.2 Semaphores

As state previously, a NUXI semaphore is an implementation of the Dijkstra’s generalized semaphore [rif!]. It is represented by the structure `t_semaphore` (lines 34–37, Section 10.4) composed of an integer field, which holds the semaphore value, and a wait channel to hold the tasks waiting for being able to decrement the value. According to Dijkstra theory, a semaphore value is always greather than or equal to zero, and each task, trying to perform a decrement, when this value is yet equal to zero, must be suspended until the semaphore itself will be incremented. Two

functions are provided, both taking a `t_semaphore` pointer as argument: `semaphore_down()`, which decrements the semaphore value suspending the task if this value would go below zero, and `semaphore_up()`, which increments the semaphore value waking up all tasks waiting for this value to become greather than zero. Since these operations must be atomic, they are performed with interrupts disabled. The implementation of these functions is the same to that of Dijkstra and is quite simple. Function `semaphore_down()` (lines 77–84, Section 10.5) tests the semaphore and sleeps the calling task until the value is different than zero; only in this last case, the semaphore value is decremented. On the other hand, `semaphore_up()` (lines 69–75, Section 10.5) increments the semaphore value and wakes up all waiting tasks associated to this semaphore (if any).

When using these routines, the `t_semaphore` pointer, passed as parameter, needs to be an initialized, i.e. the wait channel list has to be empty and the value has to be set up accordingly to program requirements (for example, if the semaphore is used as a mutex, its initial value must be one). To this aim, several macros are provided in file "wait.h" which can be used to perform semaphore initialization. In particular, macros `INIT_NEW_SEMAPHORE(n)` and `INIT_NEW_SEMAPHORE0` (lines 44–45, Section 10.4) can be used in a declaration to initialize a semaphore respectively to a value "n" and to zero. For example, if we want to declare and initialize a mutex, we can use the following code:

```
t_semaphore mutex = INIT_NEW_SEMAPHORE(1);
```

Macro `NEW_SEMAPHORE(s,n)` can be instead used to initialize runtime a semaphore "s" to value "n", for example:

```
t_semaphore mutex;
void myfunction()
{
  ...
  NEW_SEMAPHORE(mutex,1);
 ...
}
```

10.3 Condition Variables

Condition variables are used to perform entering in a critical section if a certain condition holds. For example, in the classical producer-consumer problem, the producer can put the item (and thus enter the critical section) if the shared buffer is empty (the condition). Therefore, since conditions are associated to critical section, the condition variable handling is performed using also a binary semaphore (mutex) that controls access to the critical section.

As reported in the source code (lines ??–??, Section 10.4), a NUXI condition variable is a structure composed of a wait channel, which is used to handle the sleep and wakeup mechanism, and a semaphore pointer, which holds the reference to the semaphore used to control the critical section.

Two functions are provided: `cond_wait()` and `cond_signal()` (lines ??-??, Section 10.5). The former takes, as parameters, the condition variable and the semaphore; since this function is called within a critical section, it first unlock the semaphore (by incrementing its value and waking up associated sleeping tasks) to release the critical section, and then sleeps using the wait channel. When the condition no longer holds (signalled by a call to function `cond_signal()`), it enters again in the critical section by down'ing the semaphore.

Like semaphores, two macros are provided for condition variable initialization. Macro INIT_NEW_COND is used in condition variable declaration, while NEW_COND allows intialization during execution. The following example shows the use of these macros:

```
t_cond cond = INIT_NEW_COND();

void myfunction()
{
  t_cond cond2;
  ...
  NEW_COND(cond2);
 ...
}
```

10.4 "wait.h" Header File Source

```
 1: /*
 2:  * wait.h
 3:  * Copyright (C) 2001,2002 Corrado Santoro (csanto@diit.unict.it)
 4:  *
 5:  *   The contents of this file are subject to the GNU Public License
 6:  *   (the "License"); you may not use this file except in compliance with
 7:  *   the License. You may obtain a copy of the License at http://www.fsf.org.
 8:  *
 9:  *   Software distributed under the License is distributed on an "AS IS"
10:  *   basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the
11:  *   License for the specific language governing rights and limitations
12:  *   under the License.
13:  *
14:  *   The Original Code is NUXI code.
15:  *
16:  *   The Initial Developer of the Original Code is Corrado Santoro.
```

```
17:  *   Portions created by Corrado Santoro are Copyright (C) 2001,2002.
18:  *   All Rights Reserved.
19:  */
20:
21: #ifndef __WAIT_H
22: #define __WAIT_H
23:
24: #include <kernel/ntypes.h>
25: #include <kernel/task.h>
26:
27: typedef struct wait_list {
28:   task_struct * task;
29:   struct wait_list * next;
30: } wait_list;
31:
32: typedef wait_list * wait_chan;
33:
34: typedef struct {
35:   int value;
36:   wait_chan wchan;
37: } t_semaphore;
38:
39: typedef struct {
40:   wait_chan wchan;
41:   t_semaphore * sem;
42: } t_cond;
43:
44: #define INIT_NEW_SEMAPHORE(n) { n, NULL }
45: #define INIT_NEW_SEMAPHORE0 INIT_NEW_SEMAPHORE(0)
46: #define NEW_SEMAPHORE(s,n) { s.value = n; s.wchan = NULL; }
47:
48: #define INIT_NEW_COND       { NULL , NULL }
49: #define NEW_COND(c)         { c.wchan = NULL; c.sem = NULL; }
50:
51: void thread_wakeup(wait_chan * wchan);
52: void thread_sleep(wait_chan * wchan);
53: void semaphore_up(t_semaphore * sem);
54: void semaphore_down(t_semaphore * sem);
55: void cond_wait(t_cond * cond,t_semaphore * sem);
56: void cond_signal(t_cond * cond);
57:
58: #endif
```

10.5 Concurrency Management Source Code

```
1: /*
```

```
 2:  * wait.c
 3:  * Copyright (C) 2001,2002 Corrado Santoro (csanto@diit.unict.it)
 4:  *
 5:  *   The contents of this file are subject to the GNU Public License
 6:  *   (the "License"); you may not use this file except in compliance with
 7:  *   the License. You may obtain a copy of the License at http://www.fsf.org.
 8:  *
 9:  *   Software distributed under the License is distributed on an "AS IS"
10:  *   basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the
11:  *   License for the specific language governing rights and limitations
12:  *   under the License.
13:  *
14:  *   The Original Code is NUXI code.
15:  *
16:  *   The Initial Developer of the Original Code is Corrado Santoro.
17:  *   Portions created by Corrado Santoro are Copyright (C) 2001,2002.
18:  *   All Rights Reserved.
19:  */
20:
21: #include <kernel/wait.h>
22: #include <kernel/asm.h>
23: #include <kernel/task.h>
24: #include <kernel/mm.h>
25: #include <kernel/kprintf.h>
26:
27: void in_int_sleep(wait_chan * wchan)
28: {
29:   wait_list * p;
30:   p = (wait_list *)kmalloc(sizeof(wait_list));
31:   if (p == NULL) {
32:     kprintf("No more memory for wait list allocation\r\n");
33:     return ;
34:   }
35:   current_task->state = TASK_SLEEPING;
36:   p->task = current_task;
37:   p->next = *wchan;
38:   *wchan = p;
39:
40:   asm ( "int $0x20" ::); // do scheduler
41: }
42:
43: void thread_sleep(wait_chan * wchan)
44: {
45:   __cli();
46:   in_int_sleep(wchan);
47:   __sti();
48: }
49:
50: void in_int_wakeup(wait_chan * wchan)
```

```
51: {
52:   wait_list * p;
53:   while (*wchan != NULL)
54:   {
55:     (*wchan)->task->state = TASK_READY;
56:     p = *wchan;
57:     *wchan = p->next;
58:     kfree(p);
59:   }
60: }
61:
62: void thread_wakeup(wait_chan * wchan)
63: {
64:   __cli();
65:   in_int_wakeup(wchan);
66:   __sti();
67: }
68:
69: void semaphore_up(t_semaphore * sem)
70: {
71:   __cli();
72:   ++sem->value;
73:   in_int_wakeup(&sem->wchan);
74:   __sti();
75: }
76:
77: void semaphore_down(t_semaphore * sem)
78: {
79:   __cli();
80:   while (sem->value == 0)
81:     in_int_sleep(&sem->wchan);
82:   sem->value--;
83:   __sti();
84: }
85:
86:
87: void cond_wait(t_cond * cond,t_semaphore * sem)
88: {
89:   __cli();
90:   ++sem->value;
91:   cond->sem = sem;
92:   in_int_wakeup(&sem->wchan);
93:   in_int_sleep(&cond->wchan);
94:   while (sem->value == 0)
95:     in_int_sleep(&sem->wchan);
96:   sem->value--;
97:   __sti();
98: }
99:
```

```
100: void cond_signal(t_cond * cond)
101: {
102:   __cli();
103:   if (cond->wchan != NULL)
104:   {
105:     // someone is waiting
106:     in_int_wakeup(&cond->wchan);
107:   }
108:   __sti();
109: }
110:
```

Chapter 11

Managing Memory Space

Chapter 12

Files and File Drivers

Chapter 13

What's the meaning of ...?

In Chapter 5:

Why boot code stack address 09FDF0H is converted to 09F00H:0DF0H? Indeed this linear address could be converted into 09FD0H:00F0H or 09FDFH:0000H. But you have to remember that the stack pointer in x86 processor is referred by registers SS (the segment part) and SP (the offset part) and that the stack grows towards lower memory locations, i.e. each time a "push" or a "call" occurs, the SP register is decremented by the size of pushed data. Since when SP reaches zero (and overcomes it), the processor raises a "stack overflow" exception [controllare], the stack region for a program is from offset 0 up to initial SP. Thus, setting initial stack as 09FDFH:0000H means to have no stack space while if we use 09FD0H:00F0H we would have 240 bytes (0F0H) of stack, which could be too small. The value of 09F00H:0DF0H means 3568 bytes (0DF0H) of stack which is enough for the purpose of the boot code.

.code16 statement in assembler source code: it signals the assembler to produce code for the execution in real mode. We find this statement in the boot sector "boot.s" since its code is executed by the processor in this mode.

.code32 statement in assembler source code: it signals the assembler to produce code for the execution in protected mode. We find this statement in the kernel start code "start.s" after the instruction which switches the processor in protected mode.

.balign 0x08 statement in assembler source code: it forces the alignment of the next code to a 8-byte boundary. It is useful for 64-bit processors in order to optimize the fetch phase [spiegare meglio!]

writedot/writechar subroutines in the boot sector: they write a dot or the given char on the screen. See the "INT 10H" bullet below.

.global statement in assembler source code: ...

.rept statement in assembler source code: ...

INT 10H: this software interrupt provides the screen output services for programs running in real-mode. These services are offered by ROM BIOS. The value in register AH selects the service to be performed (e.g. character output, string output, clear screen, scroll, display mode change, etc.), while the other registers are used to pass service-specific parameters. For example, AH = 0EH selects the "write single character" service; here AL has to contain the ASCII code of the character to be displayed, BL the color attribute (bits 4-7 selects background color while bits 0-3 selects the foreground), and BH the screen page to write to (text and graphic modes may have multiple pages according to the amount of EGA/VGA memory installed, page 0 is the default). [ref!][REMOVE THIS BULLET!!!!]

INT 13H: this software interrupt provides the disk I/O services for programs running in real-mode. These services are offered by ROM BIOS. The value in register AH selects the service to be performed (e.g. sector read, sector write, disk reset, track format, etc.), while the other registers are used to pass service-specific parameters. [ref!]

In Chapter 8:

LSB: it means "Least Signficant Byte" and refers to bits 0-7 of a 16-bit word.

MSB: it means "Most Signficant Byte" and refers to bits 8-15 of a 16-bit word.