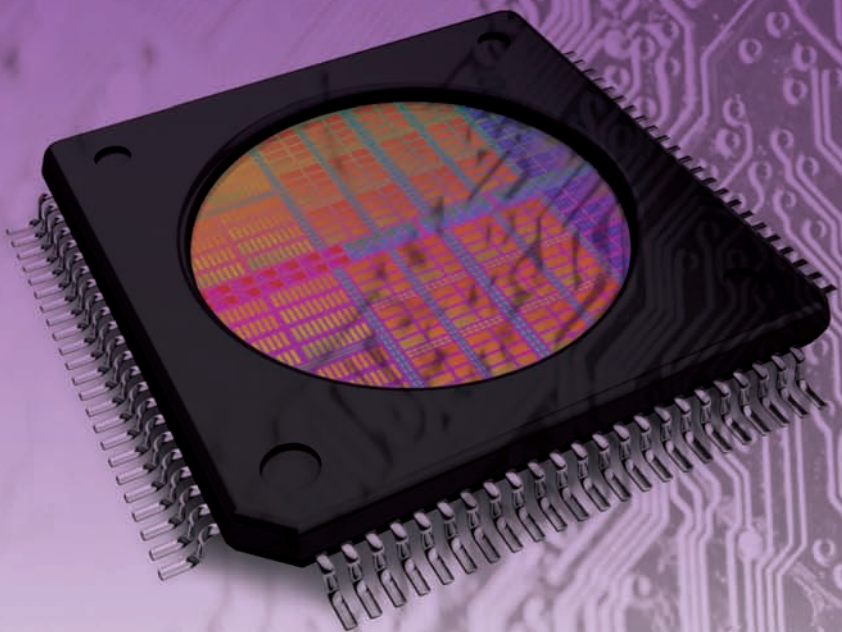
Real-Time Embedded Systems

Open-Source Operating Systems Perspective



Ivan Cibrario Bertolotti
Gabriele Manduchi

CRC Press
Taylor & Francis Group

Real-Time Embedded Systems

Open-Source Operating Systems Perspective

Embedded Systems

Series Editor

Richard Zurawski
SA Corporation, San Francisco, California, USA

Communication Architectures for Systems-on-Chip, *edited by José L. Ayala*

Real-Time Embedded Systems: Open-Source Operating Systems Perspective, *Ivan Cibrario Bertolotti and Gabriele Manduchi*

Time-Triggered Communication, *edited by Roman Obermaisser*

Real-Time Embedded Systems

Open-Source Operating Systems Perspective

Ivan Cibrario Bertolotti
Gabriele Manduchi

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20111207

International Standard Book Number-13: 978-1-4398-4161-7 (eBook - PDF)

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Foreword

Real-time embedded systems have become an integral part of our technological and social space. But is the engineering profession equipped with the right knowledge to develop those systems in ways dictated by the economic and safety aspects? Likely yes. But the knowledge is fragmented and scattered among different engineering disciplines and computing sciences. Seldom anyone of us has the clear picture of the whole. If so, then parts of it are at an abstract level. That poses a question whether the academic system provides education in a way holistic enough to prepare graduates to embark on the development of real-time embedded systems, frequently complex and imposing safety requirements. How many electrical and computer engineering departments offer subjects focusing on the application-dependent specialized communication networks used to connect embedded nodes in distributed real-time systems. If so, then the discussion is confined to the Controller Area Network (CAN), or sometimes FlexRay, in the context of automotive applications—usually a small unit of an embedded systems subject. (The impression might be that specialized communication networks are mostly used in automotive applications.) The requirement for the underlying network technology to provide real-time guarantees for message transmissions is central to proper functioning of real-time systems. Most of computer engineering streams teach operating systems. But real-time aspects are scantly covered. Computer science students, on the other hand, have very little, if any, exposure to the "physicality" of the real systems the real-time operating systems are intended to interact with. Does this put computer science graduates in a disadvantaged position? In the late 1990s and early 2000s, I was involved in the Sun Microsystems lead initiative to develop real-time extensions for the Java language. The working group comprised professionals mostly from industry with backgrounds largely in computing sciences. I was taken aback by the slow pace of the process. On reflection, the lack of exposure to the actual real-time systems in different application areas and their physicality was likely to be behind difficulties to identify generic functional requirements to be implemented by the intended extensions.

In the second part of 1980s, I was teaching digital control to the final year students of the electrical engineering course. The lab experiments to illustrate different control algorithms were designed around the, at that time, already antiquated Data General microNOVA MP/200 minicomputer, running one of the few real-time operating systems commercially available at that time—QNX, if I remember correctly. Showing things work was fun. But students' in-

sight into working of the whole system stopped at the system level commands of the operating systems. The mystery had to be revealed by discussing hypothetical implementations of the system level calls and interaction with the operating system kernel—of course at the expense of the digital control subject. At that time, seldom any electrical engineering curriculum had a separate subject dedicated to operating systems. Of frustration and to avoid the "black box" approach to illustrating control systems in action, I have written in C a simple multitasking real-time executive for MS-DOS-based platforms, to be run on an IBM PC (Intel 8088). Students were provided with the implementation documentation in addition to theoretical background; quite a lot of pages to study. But the reward was substantial: they were now in full "control." With the support of an enterprising post-graduate student, the executive was intended to be grown into more robust RTOS with a view for commercialization. But it was never to be. Academic life has other priorities. Around 1992, I decided to harness the MINIX operating system, which I then taught to the final-year graduate students, to run my real-time control lab experiments to illustrate control algorithms in their supporting real-time operating system environment. But soon after that came the Linux kernel.

If you are one of those professionals with the compartmented knowledge, particularly with the electrical and computer engineering or software engineering background, with not much theoretical knowledge of and practical exposure to real-time operating systems, this book is certainly an invaluable help to "close the loop" in your knowledge, and to develop an insight into how things work in the realm of real-time systems. Readers with a background in computer science will benefit from the hands-on approach, and a comprehensive overview of the aspects of control theory and signal processing relevant to the real-time systems. The book also discusses a range of advanced topics which will allow computer science professionals to stay up-to-date with the recent developments and emerging trends.

The book was written by two Italian researchers from the Italian National Research Council (CNR) actively working in the area of real-time (embedded) operating systems, with a considerable background in control and communication systems, and a history of the development of actual real-time systems. Both authors are also involved in teaching several courses related to these topics at Politecnico di Torino and University of Padova.

The book has been written with a remarkable clarity, which is particularly appreciated whilst reading the section on real-time scheduling analysis. The presentation of real-time scheduling is probably the best in terms of clarity I have ever read in the professional literature. Easy to understand, which is important for busy professionals keen to acquire (or refresh) new knowledge without being bogged down in a convoluted narrative and an excessive detail overload. The authors managed to largely avoid theoretical only presentation of the subject, which frequently affects books on operating systems. Selected concepts are illustrated by practical programming examples developed for the Linux and FreeRTOS operating systems. As the authors stated: Linux has a

potential to evolve in a fully fledged real-time operating system; FreeRTOS, on the other hand, gives a taste of an operating system for small footprint applications typical of most of embedded systems. Irrespective of the rationale for this choice, the availability of the programming examples allows the reader to develop insight in to the generic implementation issues transferrable to other real-time (embedded) operating systems.

This book is an indispensable addition to the professional library of anyone who wishes to gain a thorough understanding of the real-time systems from the operating systems perspective, and to stay up to date with the recent trends and actual developments of the open-source real-time operating systems.

Richard Zurawski
ISA Group, San Francisco, California

This page intentionally left blank

The Authors

Ivan Cibrario Bertolotti received the Laurea degree (*summa cum laude*) in computer science from the University of Torino, Turin, Italy, in 1996. Since then, he has been a researcher with the National Research Council of Italy (CNR). Currently, he is with the Istituto di Elettronica e di Ingegneria dell'Informazione e delle Telecomunicazioni (IEIIT), Turin, Italy.

His research interests include real-time operating system design and implementation, industrial communication systems and protocols, and formal methods for vulnerability and dependability analysis of distributed systems. His contributions in this area comprise both theoretical work and practical applications, carried out in cooperation with leading Italian and international companies.

He has taught several courses on real-time operating systems at Politecnico di Torino, Turin, Italy, starting in 2003, as well as a PhD degree course at the University of Padova in 2009. He regularly serves as a technical referee for the main international conferences and journals on industrial informatics, factory automation, and communication. He has been an IEEE member since 2006.

Gabriele Manduchi received the Laurea degree (*summa cum laude*) in electronic engineering from the University of Padova, Padua, Italy, in 1987. Since 1998 he has been a researcher with the National Research Council of Italy (CNR), and currently he is senior researcher at the Istituto Gas Ionizzati (IGI), Padua, Italy, where he leads the control and data acquisition group of the RFX nuclear fusion experiment.

His research interests include data acquisition and real-time control in large physics experiments. He is a coauthor of a software framework for data acquisition widely used in many nuclear fusion experiments around the world, and he is involved in several international projects for the management of computer infrastructures in fusion research.

He has taught several courses on computer architectures and software engineering at the University of Padova, Padua, Italy, starting in 1996, as well as PhD degree courses at the University of Padova in 2008–2010.

This page intentionally left blank

Acknowledgments

This book is the outcome of more than 10 years of research and teaching activity in the field of real-time operating systems and real-time control systems. During this time, we have been positively influenced by many other people we came in contact with, both from academy and industry. They are too numerous to mention individually, but we are nonetheless indebted to them for their contribution to our professional growth.

A special thank you goes to our university students, who first made use of the lecture notes this book is based upon. Their questions, suggestions, and remarks were helpful to make the book clearer and easier to read. In particular, we would like to thank Antonio Barbalace for his advices about Linux internals.

We would also like to express our appreciation to our coworkers for their support and patience while we were busy with the preparation of the manuscript. A special mention goes to one of Ivan's past teachers, Albert Werbrouck, who first brought his attention to the wonderful world of embedded systems.

Last, but not least, we are grateful to Richard Zurawski, who gave us the opportunity to write this book. We are also indebted to the CRC Press publishing and editorial staff: Nora Konopka, Jim McGovern, Laurie Schlags, and Jessica Vakili. Without their help, the book would probably not exist.

This page intentionally left blank

To Maria Cristina, Samuele, and Guglielmo
— ICB

To Ornella, Silvia, and Laura
— GM

This page intentionally left blank

List of Figures

List of Tables

Contents

This page intentionally left blank

1

Introduction

This book addresses three different topics: *Embedded Systems*, *Real-Time Systems*, and *Open Source Operating Systems*. Even if every single topic can well represent the argument of a whole book, they are normally intermixed in practical applications. This is in particular true for the first two topics: very often industrial or automotive applications, implemented as embedded systems, must provide timely responses in order to perform the required operation. Further, in general, real-time requirements typically refer to applications that are expected to react to the events of some kind of controlled process.

Often in the literature, real-time embedded systems are presented and analyzed in terms of abstract concepts such as tasks, priorities, and concurrence. However, in order to be of practical usage, such concepts must be then eventually implemented in *real* programs, interacting with *real* operating systems, to be executed for the control of *real* applications.

Traditionally, textbooks concentrate on specific topics using different approaches. Scheduling theory is often presented using a formal approach based on a set of assumptions for describing a computer system in a mathematical framework. This is fine, provided that the reader has enough experience and skills to understand how well real systems fit into the presented models, and this may not be the case when the textbook is used in a course or, more in general, when the reader is entering this area as a primer. Operating system textbooks traditionally make a much more limited usage of mathematical formalism and take a more practical approach, but often lack practical programming examples in the main text (some provide specific examples in appendices), as the presented concepts apply to a variety of real world systems.

A different approach is taken here: after a general presentation of the basic concepts in the first chapters, the remaining ones make explicit reference to two specific operating systems: *Linux* and *FreeRTOS*. Linux represents a full-fledged operating system with a steadily growing user base and, what is more important from the perspective of this book, is moving toward real-time responsiveness and is becoming a feasible choice for the development of real-time applications. FreeRTOS represents somewhat the opposite extreme in complexity. FreeRTOS is a minimal system with a very limited footprint in system resources and which can therefore be used in very small applications such as microcontrollers. At the same time, FreeRTOS supports a multithreading programming model with primitives for thread synchronization that are not far from what larger systems offer. If, on the one side, the choice of two

specific case studies may leave specific details of other widespread operating systems uncovered, on the other one it presents to the reader a complete conceptual path from general concepts of concurrence and synchronization down to their specific implementation, including dealing with the unavoidable idiosyncrasies of specific application programming interfaces. Here, code examples are not collected in appendices, but presented in the book chapters to stress the fact that concepts cannot be fully grasped unless undertaking the "dirty job" of writing, debugging, and running programs.

The same philosophy has been adopted in the chapters dealing with scheduling theory. It is not possible, of course, to get rid of some mathematical formalism, nor to avoid mathematical proofs (which can, however, be skipped without losing the main conceptual flow). However, thanks to the fact that such chapters follow the presentation of concurrence-related issues in operating systems, it has been possible to provide a more practical perspective to the presented results and to better describe how the used formalism maps onto real-world applications.

This book differs from other textbooks in two further aspects:

- The presentation of a case study at the beginning of the book, rather than at its end. This choice may sound bizarre as case studies are normally used to summarize presented concepts and results. However, the purpose of the case study here is different: rather than providing a final example, it is used to summarize prerequisite concepts on computer architectures that are assumed to be known by the reader afterwards. Readers may in fact have different backgrounds: less experienced ones may find the informal description of computer architecture details useful to understand more in-depth concepts that are presented later in the book such as task context switch and virtual memory issues. The more experienced will likely skip details on computer input/output or memory management, but may nevertheless have some interest in the presented application, handling online image processing over a stream of frames acquired by a digital camera.

- The presentation of the basic concepts of control theory and Digital Signal Processing in a nutshell. Traditionally, control theory and Digital Signal Processing are not presented in textbooks dealing with concurrency and schedulability, as this kind of knowledge is not strictly related to operating systems issues. However, the practical development of embedded systems is often not restricted to the choice of the optimal operating system architecture and task organization, but requires also analyzing the system from different perspectives, finding proper solutions, and finally implementing them. Different engineering disciplines cover the various facets of embedded systems: control engineers develop the optimal control strategies in the case the embedded system is devoted to process control; electronic engineers will develop the front-end electronics, such as sensor and actuator circuitry, and finally software engineers will define the computing architecture and implement the control and supervision algorithms. Ac-

tive involvement of different competencies in the development of a control or monitoring system is important in order to reduce the risk of missing major functional requirements or, on the opposite end, of an overkill, that is ending in a system which is more expensive than what is necessary. Not unusual is the situation in which the system proves both incomplete in some requirements and redundant in other aspects.

Even if several competencies may be required in the development of embedded systems, involving specialists for every system aspect is not always affordable. This may be true with small companies or research groups, and in this case different competencies may be requested by the same developer. Even when this is not the case (e.g., in large companies), a basic knowledge of control engineering and electronics is desirable for those software engineers involved in the development of embedded systems. Communication in the team can in fact be greatly improved if there is some overlap in competencies, and this may reduce the risk of flaws in the system due to the lack of communication within the development team. In large projects, different components are developed by different teams, possibly in different companies, and clear interfaces must be defined in the system's architecture to allow the proper component integration, but it is always possible that some misunderstanding could occur even with the most accurate interface definition. If there is no competence overlap among development teams, this risk may become a reality, as it happened in the development of the trajectory control system of the NASA Mars Climate Orbiter, where a software component developed by an external company was working in pounds force, while the spacecraft expected values in newtons. As a result, the $125 million Mars probe miserably crashed when it reached the Mars atmosphere [69].

As a final remark, in the title an explicit reference is made to open source systems, and two open source systems are taken as example through the book. This choice should not mislead the reader in assuming that open source systems are the common solution in industrial or automotive applications. Rather, the usage of these systems is yet limited in practice, but it is the authors' opinion that open source solutions are going to share a larger and larger portion of applications in the near future.

The book is divided into three parts: *Concurrent Programming Concepts*, *Real-Time Scheduling Analysis*, and *Advanced Topics*. The first part presents the basic concepts about processes and synchronization, and it is introduced by a case study represented by a nontrivial application for vision-based control. Along the example, the basic concepts of computer architectures and in particular of input/output management are introduced, as well as the terminology used in the rest of the book.

After the case study presentation, the basic concepts of concurrent programming are introduced. This is done in two steps: first, the main concepts are presented in a generic context without referring to any specific platform

and therefore without detailed code examples. Afterwards, the same concepts are described, with the aid of several code examples, in the context of the two reference systems: Linux and FreeRTOS.

Part I includes a chapter on network communication; even if not explicitly addressing network communication, a topic which deserves by itself a whole book, some basic concepts about network concepts and network programming are very often required when developing embedded applications.

The chapters of this part are the following:

- Chapter 2: A Case Study: Vision Control. Here, an application is presented that acquires a stream of images from a Web camera and detects online the center of a circular shape in the acquired images. This represents a complete example of an embedded application. Both theoretical and practical concepts are introduced here, such as the input/output architecture in operating systems and the video capture application programming interface for Linux.

- Chapter 3: Real-Time Concurrent Programming Principles. From this chapter onwards, an organic presentation of concurrent programming concepts is provided. Here, the concept of parallelism and its consequences, such as race conditions and deadlocks, are presented. Some general implementation issues of multiprocessing, such as process context and states, are discussed.

- Chapter 4: Deadlock. This chapter focuses on deadlock, arguably one of the most important issues that may affect a concurrent application. After defining the problem in formal terms, several solutions of practical interest are presented, each characterized by a different trade-off between ease of application, execution overhead, and conceptual complexity.

- Chapter 5: Interprocess Communication Based on Shared Variables. The chapter introduces the notions of Interprocess Communication (IPC), and it concentrates on the shared memory approach, introducing the concepts of lock variable, mutual exclusion, semaphore and monitors, which represent the basic mechanisms for process coordination and synchronization in concurrent programming.

- Chapter 6: Interprocess Communication Based on Message Passing. An alternate way for achieving interprocess communication, based on the exchange of messages, is discussed in this chapter. As in the previous two chapters, the general concepts are presented and discussed without any explicit reference to any specific operating system.

- Chapter 7: Interprocess Communication Primitives in POSIX/Linux. This chapter introduces several examples showing how the general concurrent programming concepts presented before are then mapped into Linux and POSIX. The presented information lies somewhere between a user guide and a reference for Linux/POSIX IPC primitives.

- Chapter 8: Interprocess Communication Primitives in FreeRTOS. The chapter presents the implementation of the above concurrent programming concepts in FreeRTOS, the other reference operating system for this book. The same examples of the previous chapter are used, showing how the general concepts presented in Chapters 3–6 can be implemented both on a full-fledged system and a minimal one.

- Chapter 9: Network Communication. Although not covering concepts strictly related to concurrent programming, this chapter provides important practical concepts for programming network communication using the socket abstraction. Network communication also represents a possible implementation of the message-passing synchronization method presented in Chapter 6. Several examples are provided in the chapter: although they refer to Linux applications, they can be easily ported to other systems that support the socket programming layer, such as FreeRTOS.

- Chapter 10: Lock and Wait-Free Communication. The last chapter of Part I outlines an alternative approach in the development of concurrent programs. Unlike the more classic methods discussed in Chapters 5 and 6, lock and wait-free communication never forces any participating process to wait for another. In this way, it implicitly addresses most of the problems lock-based process interaction causes to real-time scheduling—to be discussed in Chapter 15—at the expense of a greater design and implementation complexity. This chapter is based on more formal grounds than the other chapters of Part I, but it is completely self-contained. Readers not mathematically inclined can safely skip it and go directly to Part II.

The second part, Real-Time Scheduling Analysis, presents several theoretical results that are useful in practice for building systems which are guaranteed to respond within a maximum, given delay. It is worth noting now that "real-time" does not always mean "fast." Rather, a real-time system is a system whose timely response can be trusted, even if this may imply a reduced overall throughput. This part introduces the terminology and the main results in scheduling theory. They are initially presented using a simplified model which, if on the one side it allows the formal derivation of many useful properties, on the other it is still too far from real-world applications to use the above results as they are. The last two chapters of this part will progressively extend the model to include facts occurring in real applications, so that the final results can be used in practical applications.

The chapters of this part are the following:

- Chapter 11: Real-Time Scheduling Based on Cyclic Executive. This chapter introduces the basic concepts and the terminology used thorough the second part of the book. In this part, the concepts are presented in a more general way, assuming that the reader, after reading the first part of the book, is now able to use the generic concepts presented here in practi-

cal systems. A first and simple approach to real-time scheduling, cyclic executive, is presented here and its implications discussed.

- Chapter 12: Real-Time, Task-Based Scheduling. After introducing the general concepts and terminology, this chapter addresses real-time issues in the concurrent multitask model, widely described in the first part. The chapter presents two important results with immediate practical consequences: Rate Monotonic (RM) and Earliest Deadline First (EDF), which represent the optimal scheduling for fixed and variable task priority systems, respectively.

- Chapter 13: Schedulability Analysis Based on Utilization. While the previous chapter presented the optimal policies for real-time scheduling, this chapter addresses the problem of stating whether a given set of tasks can be schedulable under real-time constraints. The outcome of this chapter is readily usable in practice for the development of real-time systems.

- Chapter 14: Schedulability Analysis Based on Response Time Analysis. This chapter provides a refinement of the results presented in the previous one. Although readily usable in practice, the results of Chapter 13 provide a conservative approach, which can be relaxed with the procedures presented in this chapter, at the cost of a more complex schedulability analysis. The chapter also takes into account sporadic tasks, whose behavior cannot be directly described in the general model used so far, but which nevertheless describe important facts, such as the occurrence of exceptions that happen in practice.

- Chapter 15: Process Interactions and Blocking. This chapter and the next provide the concepts that are required to map the theoretical results on scheduling analysis presented up to now onto real-world applications, where the tasks cannot anymore be described as independent processes, but interact with each other. In particular, this chapter addresses the interference among tasks due to the sharing of system resources, and introduces the priority inheritance and priority ceiling procedures, which are of fundamental importance in the implementation of real-world applications.

- Chapter 16: Self-Suspension and Schedulability Analysis. This chapter addresses another fact which differentiates real systems from the model used to derive the theoretical results in schedulability analysis, that is, the suspension of tasks due, for instance, to I/O operations. The implications of this fact, and the quantification of its effects, are discussed here.

The last part will cover other aspects of embedded systems. Unlike the first two parts, where concepts are introduced step by step to provide a comprehensive understanding of concurrent programming and real-time systems, the chapters of the last part cover separate, self-consistent arguments. The chapters of this part are the following:

- Chapter 17: Internal Structure of FreeRTOS. This chapter gives a description of the internals of FreeRTOS. Thanks to its simplicity, it has been possible to provide a sufficiently detailed example showing how the concurrent programming primitives are implemented in practice. The chapter provides also practical indications on how the system can be ported to new architectures.

- Chapter 18: Internal Structures and Operating Principles of Linux Real-Time Extensions. It is of course not possible to provide in a single chapter a detailed description of the internals of a complex system, such as Linux. Nevertheless, this chapter will illustrate the main ideas and concepts in the evolution of a general purpose operating system, and in particular of Linux, towards real-time responsiveness.

- Chapter 19: OS Abstraction Layer. This chapter addresses issues that are related to software engineering, and presents an object-oriented approach in the development of multiplatform applications. Throughout the book general concepts have been presented, and practical examples have been provided, showing that the same concepts are valid on different systems, albeit using a different programming interface. If an application has to be implemented for different platforms, it is convenient, therefore, to split the code in two parts, moving the semantics of the program in the platform-independent part, and implementing a common abstraction of the underlying operating system in the other system-dependent one.

- Chapter 20: Basics of Control Theory and Digital Signal Processing. This chapter provides a quick tour of the most important mathematical concepts for control theory and digital signal processing, using two case studies: the control of a pump and the development of a digital low-pass filter. The only mathematical background required of the reader corresponds to what is taught in a base math course for engineering, and no specific previous knowledge in control theory and digital signal processing is assumed.

The short bibliography at the end of the book has been compiled with less experienced *readers* in mind. For this reason, we did not provide an exhaustive list of references, aimed at acknowledging each and every author who contributed to the rather vast field of real-time systems.

Rather, the bibliography is meant to point to a limited number of additional sources of information, which readers can and should actually use as a starting point to seek further information, without getting lost. There, readers will also find more, and more detailed, references to continue their quest.

Part I

Concurrent Programming Concepts

This page intentionally left blank

2

A Case Study: Vision Control

CONTENTS

This chapter describes a case study consisting of an embedded application performing online image processing. Both theoretical and practical concepts are introduced here: after an overview of basic concepts in computer input/output, some important facts on operating systems (OS) and software complexity will be presented here. Moreover, some techniques for software optimization and parallelization will be presented and discussed in the framework of the presented case study. The theory and techniques that are going to be introduced do not represent the main topic of this book. They are necessary, nevertheless, to fully understand the remaining chapters, which will concentrate on more specific aspects such as multithreading and process scheduling.

The presented case study consists of a Linux application that acquires a sequence of images (frames) from a video camera device. The data acquisition program will then perform some elaboration on the acquired images in order to detect the coordinates of the center of a circular shape in the acquired images.

This chapter is divided into four main sections. In the first section general concepts in computer input/output (I/O) are presented. The second section will discuss how I/O is managed by operating systems, in particular Linux,

while in the third one the implementation of the frame acquisition is presented. The fourth section will concentrate on the analysis of the acquired frames to retrieve the desired information; after presenting two widespread algorithms for image analysis, the main concepts about software complexity will be presented, and it will be shown how the execution time for those algorithms can be reduced, sometimes drastically, using a few optimization and parallelization techniques.

Embedded systems carrying out online analysis of acquired images are becoming widespread in industrial control and surveillance. In order to acquire the sequence of the frames, the video capture application programming interface for Linux (V4L2) will be used. This interface supports most commercial USB webcams, which are now ubiquitous in laptops and other PCs. Therefore this sample application can be easily reproduced by the reader, using for example his/her laptop with an integrated webcam.

2.1 Input Output on Computers

Every computer does input/output (I/O); a computer composed only of a processor and the memory would do barely anything useful, even if containing all the basic components for running programs. I/O represents the way computers interact with the outside environment. There is a great variety of I/O devices: A personal computer will input data from the keyboard and the mouse, and output data to the screen and the speakers while using the disk, the network connection, and the USB ports for both input and output. An embedded system typically uses different I/O devices for reading data from sensors and writing data to actuators, leaving user interaction be handled by remote clients connected through the local area network (LAN).

2.1.1 Accessing the I/O Registers

In order to communicate with I/O devices, computer designers have followed two different approaches: *dedicated I/O bus* and *memory-mapped I/O*. Every device defines a set of registers for I/O management. *Input registers* will contain data to be read by the processor; *output registers* will contain data to be outputted by the device and will be written by the processor; *status registers* will contain information about the current status of the device; and finally *control registers* will be written by the processor to initiate or terminate device activities.

When a dedicated bus is defined for the communication between the processor and the device registers, it is also necessary that specific instructions for reading or writing device register are defined in the set of machine instructions. In order to interact with the device, a program will read and write appropriate

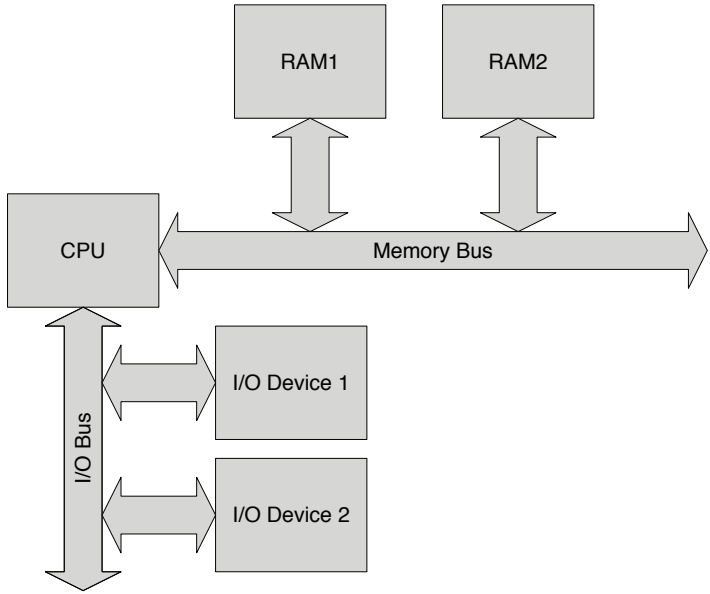


FIGURE 2.1
Bus architecture with a separate I/O bus.

values onto the I/O bus locations (i.e., at the addresses corresponding to the device registers) via specific I/O Read and Write instructions.
In memory-mapped I/O, devices are seen by the processor as a set of registers, but no specific bus for I/O is defined. Rather, the same bus used to exchange data between the processor and the memory is used to access I/O devices. Clearly, the address range used for addressing device registers must be disjoint from the set of addresses for the memory locations. Figure 2.1 and Figure 2.2 show the bus organization for computers using a dedicated I/O bus and memory-mapped I/O, respectively. Memory-mapped architectures are more common nowadays, but connecting all the external I/O devices directly to the memory bus represents a somewhat simplified solution with several potential drawbacks in reliability and performance. In fact, since speed in memory access represents one of the major bottlenecks in computer performance, the memory bus is intended to operate at a very high speed, and therefore it has very strict constraints on the electrical characteristics of the bus lines, such as capacity, and in their dimension. Letting external devices be directly connected to the memory bus would increase the likelihood that possible malfunctions of the connected devices would seriously affect the function of the whole system and, even if that were not the case, there would be the concrete risk of lowering the data throughput over the memory bus.
In practice, one or more separate buses are present in the computer for I/O, even with memory-mapped architectures. This is achieved by letting a *bridge*

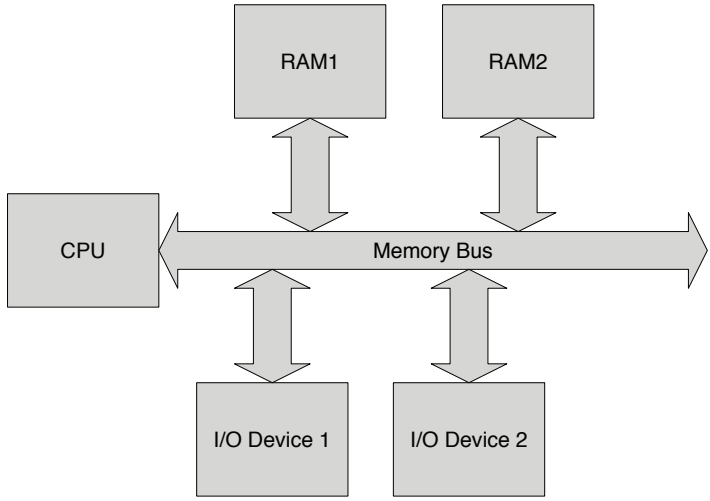


FIGURE 2.2
Bus architecture for Memory Mapped I/O.

component connect the memory bus with the I/O bus. The bridge presents itself to the processor as a device, defining a set of registers for programming the way the I/O bus is mapped onto the memory bus. Basically, a bridge can be programmed to define one or more address mapping windows. Every address mapping window is characterized by the following parameters:

1. Start and end address of the window in the memory bus
2. Mapping address offset

Once the bridge has been programmed, for every further memory access performed by the processor whose address falls in the selected address range, the bridge responds in the bus access protocol and translates the read or write operation performed in the memory bus into an equivalent read or write operation in the I/O bus. The address used in the I/O bus is obtained by adding the preprogrammed address offset for that mapping window. This simple mechanism allows to decouple the addresses used by I/O devices over the I/O bus from the addresses used by the processor.

A common I/O bus in computer architectures is the Peripheral Component Interconnect (PCI) bus, widely used in personal computers for connecting I/O devices. Normally, more than one PCI segment is defined in the same computer board. The PCI protocol, in fact, poses a limit in the number of connected devices and, therefore, in order to handle a larger number of devices, it is necessary to use PCI to PCI bridges, which connect different segments of the PCI bus. The bridge will be programmed in order to define map address windows in the primary PCI bus (which sees the bridge as a device connected to the

bus) that are mapped onto the corresponding address range in the secondary PCI bus (for which the bridge is the master, i.e., leads bus operations). Following the same approach, new I/O buses, such as the Small Computer System Interface (SCSI) bus for high-speed disk I/O, can be integrated into the computer board by means of bridges connecting the I/O bus to the memory bus or, more commonly, to the PCI bus. Figure 2.3 shows an example of bus configuration defining a memory to PCI bridge, a PCI to PCI bridge, and a PCI to SCSI bridge.

One of the first actions performed when a computer boots is the configuration of the bridges in the system. Firstly, the bridges directly connected to the memory bus are configured, so that the devices over the connected buses can be accessed, including the registers of the bridges connecting these to new I/O buses. Then the bridges over these buses are configured, and so on. When all the bridges have been properly configured, the registers of all the devices in the system are directly accessible by the processor at given addresses over the memory bus. Properly setting all the bridges in the system may be tricky, and a wrong setting may make the system totally unusable. Suppose, for example, what could happen if an address map window for a bridge on the memory bus were programmed with an overlap with the address range used by the RAM memory. At this point the processor would be unable to access portions of memory and therefore would not anymore be able to execute programs.

Bridge setting, as well as other very low-level configurations are normally performed before the operating system starts, and are carried out by the Basic Input/Output System (BIOS), a code which is normally stored on ROM and executed as soon as the computer is powered. So, when the operating system starts, all the device registers are available at proper memory addresses. This is, however, not the end of the story: in fact, even if device registers are seen by the processor as if they were memory locations, there is a fundamental difference between devices and RAM blocks. While RAM memory chips are expected to respond in a time frame on the order of nanoseconds, the response time of devices largely varies and in general can be much longer. It is therefore necessary to synchronize the processor and the I/O devices.

2.1.2 Synchronization in I/O

Consider, for example, a serial port with a baud rate of 9600 bit/s, and suppose that an incoming data stream is being received; even if ignoring the protocol overhead, the maximum incoming byte rate is 1200 byte/s. This means that the computer has to wait 0.83 milliseconds between two subsequent incoming bytes. Therefore, a sort of synchronization mechanism is needed to let the computer know when a new byte is available to be read in a data register for readout. The simplest method is *polling*, that is, repeatedly reading a status register that indicates whether new data is available in the data register. In this way, the computer can synchronize itself with the actual data rate of the

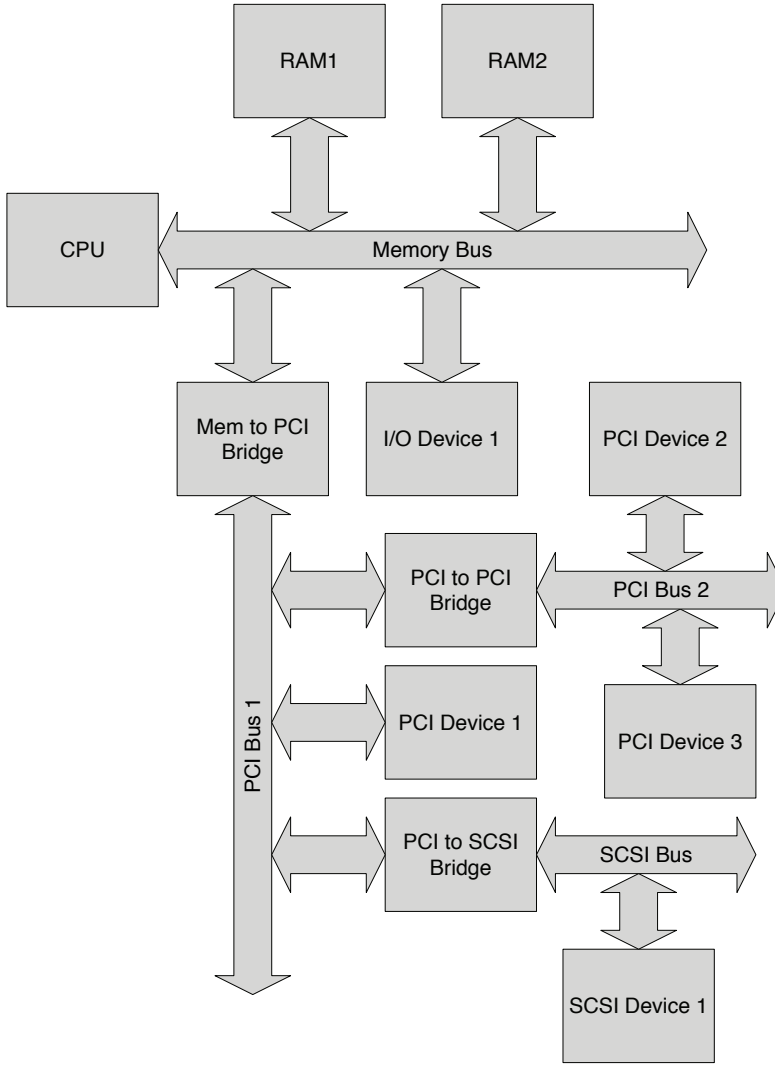


FIGURE 2.3
Bus architecture with two PCI buses and one SCSI bus.

device. This comes, however, at a cost: no useful operation can be carried out by the processor when synchronizing to devices in polling. If we assume that 100 ns are required on average for memory access, and assuming that access to device registers takes the same time as a memory access (a somewhat simplified scenario since we ignore here the effects of the memory cache), acquiring a data stream from the serial port would require more than 8000 read operations of the status register for every incoming byte of the stream – that is, wasting 99.99% of the processor power in useless accesses to the status register. This situation becomes even worse for slower devices; imagine the percentage of processor power for doing anything useful if polling were used to acquire data from the keyboard!

Observe that the operations carried out by I/O devices, once programmed by a proper configuration of the device registers, can normally proceed in parallel with the execution of programs. It is only required that the device should notify the processor when an I/O operation has been completed, and new data can be read or written by the processor. This is achieved using Interrupts, a mechanism supported by most I/O buses. When a device has been started, typically by writing an appropriate value in a command register, it proceeds on its own. When new data is available, or the device is ready to accept new data, the device raises an interrupt request to the processor (in most buses, some lines are dedicated to interrupt notification) which, as soon as it finishes executing the current machine instruction, will serve the interrupt request by executing a specific routine, called Interrupt Service Routine (ISR), for the management of the condition for which the interrupt has been generated.

Several facts must be taken into account when interrupts are used to synchronize the processor and the I/O operations. First of all, more than one device could issue an interrupt at the same time. For this reason, in most systems, a priority is associated with interrupts. Devices can in fact be ranked based on their importance, where important devices require a faster response. As an example, consider a system controlling a nuclear plant: An interrupt generated by a device monitoring the temperature of a nuclear reactor core is for sure more important than the interrupt generated by a printer device for printing daily reports. When a processor receives an interrupt request with a given associated priority level N, it will soon respond to the request only if it is not executing any service routine for a previous interrupt of priority M, $M \geq N$. In this case, the interrupt request will be served as soon as the previous Interrupt Service Routine has terminated and there are no pending interrupts with priority greater or equal to the current one.

When a processor starts serving an interrupt, it is necessary that it does not lose information about the program currently in execution. A program is fully described by the associated memory contents (the program itself and the associated data items), and by the content of the processor registers, including the Program Counter (PC), which records the address of the current machine instruction, and the Status Register (SR), which contains information on the

current processor status. Assuming that memory locations used to store the program and the associated data are not overwritten during the execution of the interrupt service routine, it is only necessary to preserve the content of the processor registers. Normally, the first actions of the routine are to save in the stack the content of the registers that are going to be used, and such registers will be restored just before its termination. Not all the registers can be saved in this way; in particular, the PC and the SR are changed just before starting the execution of the interrupt service routine. The PC will be set to the address of the first instruction of the routine, and the SR will be updated to reflect the fact that the process is starting to service an interrupt of a given priority. So it is necessary that these two register are saved by the processor itself and restored when the interrupt service routine has finished (a specific instruction to return from ISR is defined in most computer architectures). In most architectures the SR and PC registers are saved on the stack, but others, such as the ARM architecture, define specific registers to hold the saved values.

A specific interrupt service routine has to be associated with every possible source of interrupt, so that the processor can take the appropriate actions when an I/O device generates an interrupt request. Typically, computer architectures define a vector of addresses in memory, called a *Vector Table*, containing the start addresses of the interrupt service routines for all the I/O devices able to generate interrupt requests. The offset of a given ISR within the vector table is called the *Interrupt Vector Number*. So, if the interrupt vector number were communicated by the device issuing the interrupt request, the right service routine could then be called by the processor. This is exactly what happens; when the processor starts serving a given interrupt, it performs a cycle on the bus called the *Interrupt Acknowledge Cycle* (IACK) where the processor communicates the priority of the interrupt being served, and the device which issued the interrupt request at the specified priority returns the interrupt vector number. In case two different devices issued an interrupt request at the same time with the same priority, the device closest to the processor in the bus will be served. This is achieved in many buses by defining a bus line in *Daisy Chain* configuration, that is, which is propagated from every device to the next one along the bus, only in cases where it did not answer to an IACK cycle. Therefore, a device will answer to an IACK cycle only if both conditions are met:

1. It has generated a request for interrupt at the specified priority
2. It has received a signal over the daisy chain line

Note that in this case it will not propagate the daisy chain signal to the next device.

The offset returned by the device in an IACK cycle depends on the current organization of the vector table and therefore must be a programmable parameter in the device. Typically, all the devices which are able to issue an

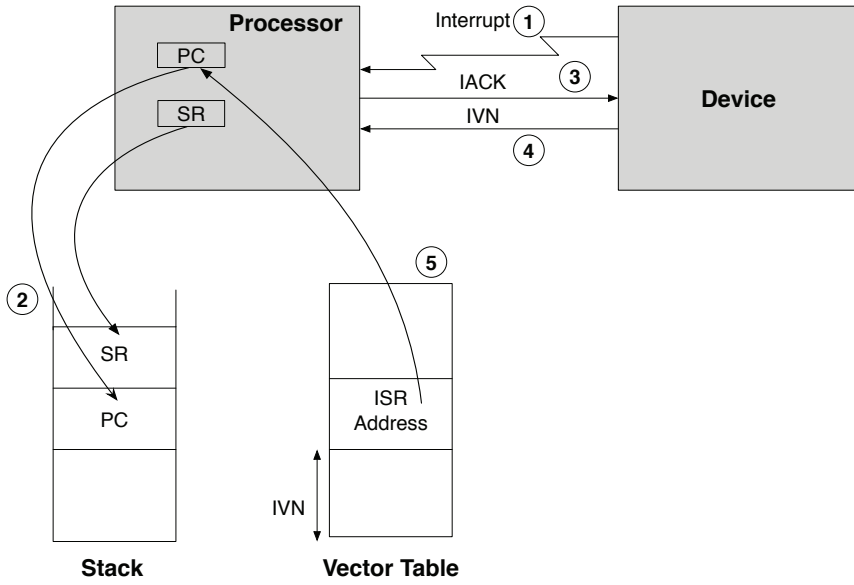


FIGURE 2.4
The Interrupt Sequence.

interrupt request have two registers for the definition of the interrupt priority and the interrupt vector number, respectively. The sequence of actions is shown in Figure 2.4, highlighting the main steps of the sequence:

1. The device issues an interrupt request;
2. The processor saves the context, i.e., puts the current values of the PC and of the SR on the stack;
3. The processor issues an interrupt acknowledge cycle (IACK) on the bus;
4. The device responds by putting the interrupt vector number (IVN) over the data lines of the bus;
5. The processor uses the IVN as an offset in the vector table and loads the interrupt service routine address in the PC.

Programming a device using interrupts is not a trivial task, and it consists of the following steps:

1. The interrupt service routine has to be written. The routine can assume that the device is ready at the time it is called, and therefore no synchronization (e.g., polling) needs to be implemented;
2. During system boot, that is when the computer and the connected

I/O devices are configured, the code of the interrupt service routine has to be loaded in memory, and its start address written in the vector table at, say, offset N;

3. The value N has to be communicated to the device, usually written in the interrupt vector number register;

4. When an I/O operation is requested by the program, the device is started, usually by writing appropriate values in one or more command registers. At this point the processor can continue with the program execution, while the device operates. As soon as the device is ready, it will generate an interrupt request, which will be eventually served by the processor by running the associated interrupt service routine.

In this case it is necessary to handle the fact that data reception is asynchronous. A commonly used techniques is to let the program continue after issuing an I/O request until the data received by the device is required. At this point the program has to suspend its execution waiting for data, unless not already available, that is, waiting until the corresponding interrupt service routine has been executed. For this purpose the interprocess communication mechanisms described in Chapter 5 will be used.

2.1.3 Direct Memory Access (DMA)

The use of interrupts for synchronizing the processor and the connected I/O devices is ubiquitous, and we will see in the next chapters how interrupts represent the basic mechanism over which operating systems are built. Using interrupts clearly spares processor cycles when compared with polling; however, there are situations in which even interrupt-driven I/O would require too much computing resources. To better understand this fact, let's consider a mouse which communicates its current position by interrupting the processor 30 times per second. Let's assume that 400 processor cycles are required for the dispatching of the interrupt and the execution of the interrupt service routine. Therefore, the number of processor cycles which are dedicated to the mouse management per second is $400 * 30 = 12000$. For a 1 GHz clock, the fraction of processor time dedicated to the management of the mouse is $12000/10^9$, that is, 0.0012% of the processor load. Managing the mouse requires, therefore, a negligible fraction of processor power.

Consider now a hard disk that is able to read data with a transfer rate of 4 MByte/s, and assume that the device interrupts the processor every time 16 bytes of data are available. Let's also assume that 400 clock cycles are still required to dispatch the interrupt and execute the associated service routine. The device will therefore interrupt the processor 250000 times per second, and 10^8 processor cycles will be dedicated to handle data transfer every second.

For a 1 GHz processor this means that 10% of the processor time is dedicated to data transfer, a percentage clearly no more acceptable.

Very often data exchanged with I/O devices are transferred from or to memory. For example, when a disk block is read it is first transferred to memory so that it is later available to the processor. If the processor itself were in charge of transferring the block, say, after receiving an interrupt request from the disk device to signal the block availability, the processor would repeatedly read data items from the device's data register into an internal processor register and write it back into memory. The net effect is that a block of data has been transferred from the disk into memory, but it has been obtained at the expense of a number of processor cycles that could have been used to do other jobs if the device were allowed to write the disk block into memory by itself. This is exactly the basic concept of *Direct Memory Access* (DMA), which is letting the devices read and write memory by themselves so that the processor will handle I/O data directly in memory. In order to put this simple concept in practice it is, however, necessary to consider a set of facts. First of all, it is necessary that the processor can "program" the device so that it will perform the correct actions, that is, reading/writing a number N of data items in memory, starting from a given memory address A. For this purpose, every device able to perform DMA provides at least the following registers:

- A Memory Address Register (MAR) initially containing the start address in memory of the block to be transferred;
- A Word Count register (WC) containing the number of data items to be transferred.

So, in order to program a block read or write operation, it is necessary that the processor, after allocating a block in memory and, in case of a write operation, filling it with the data to be output to the device, writes the start address and the number of data items in the MAR and WC registers, respectively. Afterwards the device will be started by writing an appropriate value in (one of) the command register(s). When the device has been started, it will operate in parallel with the processor, which can proceed in the execution of the program. However, as soon as the device is ready to transfer a data item, it will require the memory bus used by the processor to exchange data with memory, and therefore some sort of bus arbitration is needed since it is not possible that two devices read or write the memory at the same time on the same bus (note however that nowadays memories often provide multiport access, that is, allow simultaneous access to different memory addresses). At any time one, and only one, device (including the processor) connected to the bus is the *master*, i.e., can initiate a read or write operation. All the other connected devices at that time are slaves and can only answer to a read/write bus cycle when they are addressed. The memory will be always a slave in the bus, as well as the DMA-enabled devices when they are not performing DMA. At the time such a device needs to exchange data with the memory, it will

ask the current master (normally the processor, but it may be another device performing DMA) the ownership of the bus. For this purpose the protocol of every bus able to support ownership transfer is to define a cycle for the bus ownership transfer. In this cycle, the potential master raises a request line and the current master, in response, relinquishes the mastership, signaling this over another bus line, and possibly waiting for the termination of a read/write operation in progress. When a device has taken the bus ownership, it can then perform the transfer of the data item and will remain the current master until the processor or another device asks to become the new master. It is worth noting that the bus ownership transfers are handled by the bus controller components and are carried out entirely in hardware. They are, therefore, totally transparent to the programs being executed by the processor, except for a possible (normally very small) delay in their execution.
Every time a data item has been transferred, the MAR is incremented and the WC is decremented. When the content of the WC becomes zero, all the data have been transferred, and it is necessary to inform the processor of this fact by issuing an interrupt request. The associated Interrupt Service Routine will handle the block transfer termination by notifying the system of the availability of new data. This is normally achieved using the interprocess communication mechanisms described in Chapter 5.

2.2 Input/Output Operations and the Operating System

After having seen the techniques for handling I/O in computers, the reader will be convinced that it is highly desirable that the complexity of I/O should be handled by the operating system and not by user programs. Not surprisingly, this is the case for most operating systems, which offer a unified interface for I/O operations despite the large number of different devices, each one defining a specific set of registers and requiring a specific I/O protocol. Of course, it is not possible that operating systems could include the code for handling I/O in every available device. Even if it were the case, and the developers of the operating system succeed in the titanic effort of providing the device specific code for every known device, the day after the system release there will be tens of new devices not supported by such an operating system. For this reason, operating systems implement the generic I/O functionality, but leave the details to a device-specific code, called the *Device Driver*. In order to be integrated into the system, every device requires its software driver, which depends not only on the kind of hardware device but also on the operating system. In fact, every operating system defines its specific set of interfaces and rules a driver must adhere to in order to be integrated. Once installed, the driver becomes a component of the operating system. This means that a failure in the device driver code execution becomes a failure of the operating

system, which may lead to the crash of the whole system. (At least in monolithic operating systems such as Linux and Windows; this may be not true for other systems, such as microkernel-based ones.) User programs will never interact directly with the driver as the device is accessible only via the Application Programming Interface (API) provided by the operating system. In the following we shall refer to the Linux operating systems and shall see how a uniform interface can be adapted to the variety of available devices. The other operating systems adopt a similar architecture for I/O, which typically differ only by the name and the arguments of the I/O systems routines, but not on their functionality.

2.2.1 User and Kernel Modes

We have seen how interacting with I/O devices means reading and writing into device registers, mapped at given memory addresses. It is easy to guess what could happen if user programs were allowed to read and write at the memory locations corresponding to device registers. The same consideration holds also for the memory structures used by the operating system itself. If user programs were allowed to freely access the whole addressing range of the computer, an error in a program causing a memory access to a wrong address (something every C programmer experiences often) may lead to the corruption of the operating system data structures, or to an interference with the device operation, leading to a system crash.
For this reason most processors define at least two levels of execution: *user mode* and *kernel* (or *supervisor*) *mode*. When operating in user mode, a program is not allowed to execute some machine instructions (called *Privileged Instructions*) or to access sets of memory addresses. Conversely, when operating in kernel mode, a program has full access to the processor instructions and to the full addressing range. Clearly, most of the operating system code will be executed in kernel mode, while user programs are kept away from dangerous operations and are intended to be executed in user mode. Imagine what would happen if the HALT machine instruction for stopping the processor were available in user mode, possibly on a server with tens of connected users.

A first problem arises when considering how a program can switch from user to kernel mode. If this were carried out by a specific machine instruction, would such an instruction be accessible in user mode? If not, it would be useless, but if it were, the barrier between kernel mode and user mode would be easily circumvented, and malicious programs could easily take the whole system down.

So, how to solve the dilemma? The solution lies in a new mechanism for the invocation of software routines. In the normal routine invocation, the calling program copies the arguments of the called routine over the stack and then puts the address of the first instruction of the routine into the Program Counter register, after having copied on the stack the return address, that is, the address of the next instruction in the calling program. Once the called

routine terminates, it will pick the saved return address from the stack and put it into the Program Counter, so that the execution of the calling program is resumed. We have already seen, however, how the interrupt mechanism can be used to "invoke" an interrupt service routine. In this case the sequence is different, and is triggered not by the calling program but by an external hardware device. It is exactly when the processor starts executing an Interrupt Service routine that the current execution mode is switched to kernel mode. When the interrupt service routine returns and the interrupted program resumes its execution, unless not switching to a new interrupt service routine, the execution mode is switched to user mode. It is worth noting that the mode switch is not controlled by the software, but it is the processor which only switches to kernel mode when servicing an interrupt.

This mechanism makes sense because interrupt service routines interact with devices and are part of the device driver, that is, of a software component that is integrated in the operating system. However, it may happen that user programs have to do I/O operations, and therefore they need to execute some code in kernel mode. We have claimed that all the code handling I/O is part of the operating system and therefore the user program will call some system routine for doing I/O. However, how do we switch to kernel mode in this case where the trigger does not come from an hardware device? The solution is given by *Software Interrupts*. Software interrupts are not triggered by an external hardware signal, but by the execution of a specific machine instruction. The interrupt mechanism is quite the same: The processor saves the current context, picks the address of the associated interrupt service routine from the vector table and switches to kernel mode, but in this case the Interrupt Vector number is not obtained by a bus IACK cycle; rather, it is given as an argument to the machine instruction for the generation of the software interrupt.

The net effect of software interrupts is very similar to that of a function call, but the underlying mechanism is completely different. This is the typical way the operating system is invoked by user programs when requesting system services, and it represents an effective barrier protecting the integrity of the system. In fact, in order to let any code to be executed via software interrupts, it is necessary to write in the vector table the initial address of such code but, not surprisingly, the vector table is not accessible in user mode, as it belongs to the set of data structures whose integrity is essential for the correct operation of the computer. The vector table is typically initialized during the system boot (executed in kernel mode) when the operating system initializes all its data structures.

To summarize the above concepts, let's consider the execution story of one of the most used C library function: `printf()`, which takes as parameter the (possibly formatted) string to be printed on the screen. Its execution consists of the following steps:

1. The program calls routine `printf()`, provided by the C run time