SIRTOS: A Simple Real-Time Operating System

Vasileios Kouliaridis, Vasileios Vlachos, Ilias K. Savvas
Dept. of Computer Science and Engineering
TEI of Thessaly, Greece
billkoul8@gmail.com,{vsvlachos, savvas}@teilar.gr

Iosif Androulidakis
University of Ioannina
sandro@noc.uoi.gr

Abstract —This paper describes the development of a simple, real-time operating system for educational purposes. The system provides soft real-time capabilities to serve real time processes and data and meet deadlines. The main contribution of this paper is a very simple open source operating system that implements different real-time algorithms. The code is kept minimal as the kernel is only 15000 lines long. It supports high definition graphics and easy modification which allows more real-time algorithms to be implemented.

Keywords — real- time operating system, open source, SIRTOS;

I. INTRODUCTION

In every operating system the kernel is the central and most important module. Most operating systems are named after their kernel as it is the part of the operating system that loads first, and stays permanently in memory. Therefore, it is important for the kernel to be as small and simple as possible while still providing all the essential services. The kernel is loaded into a protected area of memory to prevent it from overwrite either by programs accidently or for malicious purposes. It is responsible for interrupt management, memory management, process management, and input/output device management.

Real-time operating systems require a more complex and advanced kernel. The kernel in a real-time operating system has to handle real-time processes and meet deadlines while at the same time to maintain system performance and stability. These types of operating systems are usually used for industrial, medical or military applications because of their quick and up to time responses.

Given a real-time system, the goal is to schedule the system's tasks on a processor, or processors, so that each task completes execution before a specified deadline **Chyba! Nenašiel sa žiaden zdroj odkazov.**. Systems that can always guarantee these deadlines are called hard real-time systems. The soft real-time systems are based on a best effort approach while firm real time systems, are those that meet the real time requirements most of the times. Deploying an airbag in response to a sensor being triggered has to be implemented in a hard real-time system. On the other, hand decoding video frames is an example of where a firm real-time system will suffice, where a general desktop personal computer is usually supported by a soft real time system.

The rest of the paper is organized as follows. The related work is presented in Section II. In Section III real systems are described. In Section IV the implementation of the proposed system is presented while in Section V the development of the system is given. Finally, Section VI concludes the paper and highlights some future research directions.

II. REAL TIME SYSTEMS

A. Real-Time Scheduling

Many different real-time scheduling algorithms have been proposed to assign priorities to tasks and processes in an operating system. Most algorithms are classified as static priority, dynamic priority, or mixed priority. A static-priority algorithm assigns all priorities at design time, and those priorities remain constant for the whole life span of the task. A dynamic-priority algorithm assigns priorities at runtime, based on execution parameters of tasks, such as upcoming deadlines. A mixed-priority algorithm has both static and dynamic components. Static-priority algorithms are usually simpler than algorithms that must compute priorities during runtime, but it is not always possible to statically map all possible scenarios in the design time [4][5].

Scheduling algorithms are also separated in two other types, preemptive and non-preemptive. In preemptive algorithms when a task is executed on a processor, if another task arrives with higher priority, it will stop whereas in non-preemptive algorithms, the task's execution doesn't stop until it completed and finished [12].

Rate monotonic (RM) is the most well-known algorithm for assigning static priorities to periodic processes. Scheduling is performed by a simple priority scheduler. At each quantum, the highest priority ready process gets to run. Processes at the same priority level run round-robin. A set of tasks scheduled under the RM scheme is schedulable if the following condition holds for every task T_i [3].

$$\exists t : 0 < t < p_i : \sum_{j=1}^{i} [\frac{t}{p_j}] \times c_j \sum_{j=1}^{i-1} [\frac{t-1}{p_j}] \times s \leq t \quad (1)$$

- N - The number of tasks in the system.
- p_j - The period of task
- c_j - The worst-case computational cost (execution time) of a task when it is the only task executing on the processor.
- s - The execution time required for one loop iteration in the implementation of a lock-free object, which for simplicity assumed to be the same for all objects [1][8].

Rate-monotonic scheduling (RMS) is one of the most widely used scheduling policies on uniprocessors for preemptive periodic real-time tasks [7].

Earliest deadline first (EDF) scheduling algorithm on the other hand is simple in concept. When selecting a task for execution, an EDF scheduling algorithm chooses the task with the earliest deadline. Ties between tasks with identical deadlines are broken arbitrarily. With a non-preemptive formulation of the EDF algorithm, once a task is selected, the task is immediately executed to completion [9].

Each process notifies the operating system scheduler its absolute time deadline. The algorithm simply allows the process that is most likely to miss its deadline to run first. Generally, this means that one process will run to completion if it has an earlier deadline than another. The only time a process would be preempted would be when a new process with an even shorter deadline becomes ready to run. A set of periodic tasks scheduled under the EDF scheme is schedulable if the following condition holds.

$$\sum_{i=1}^{N} \frac{C_i + s}{p_i} \leq 1 \quad (2)$$

Where

- N - The number of tasks in the system.
- p_i - The period of task
- c_i - The worst-case computational cost (execution time) of a task when it is the only task executing on the processor.
- s - The execution time required for one loop iteration in the implementation of a lock-free object, which for simplicity is assumed to be the same for all objects [1][12].

The EDF algorithm has been proven to be an optimal uniprocessor scheduling algorithm, also if the number of periodic tasks is large EDF is recommended [2]. If a set of tasks is unschedulable under EDF no other scheduling algorithm can feasible schedule this task set [13].

One disadvantage of EDF is that its behavior becomes unpredictable in overloaded situations and the performance of an *EDF algorithm* drops in overloaded conditions and it cannot be considered for use [6].

B. *Inter-process communication*

In most multitasking operating systems it is important to manage data sharing among tasks. If two tasks try to access the same data or hardware resources at the same time, the results could be unpredictable. Three major methods to avoid this have been proposed and are widely used. The first approach is to temporarily mask or disable the interrupts. This method has the lowest overhead to prevent multiple tasks accessing the same data. When interrupts are masked the current task is the only one using the CPU since no other task is permitted to take control and use the same section. These sections are called critical sections. Of course critical sections may block higher priority ISRs from running.

The second option is to use semaphores. Using semaphores is simple to understand and implement. A binary semaphore is either locked or unlocked. When the semaphore is locked tasks must wait until the semaphore is unlocked again.

The third and most elegant implementation is by message passing. This method works by giving control of a certain resource to only one task. If another task wants to manipulate this resource, it sends a message to the managing task.

One way to calculate the schedulability of a set of periodic tasks in dynamic priority scheduling is with the following condition.

$$DPsched = \sum_{j=1}^{N} \frac{C_j + block_j}{p_j} \leq 1 \qquad (3)$$

Where block $_i$ is the maximum time for which a task T_i can be blocked by some lower priority task and equals the time to execute the longest critical section [1].

III. IMPLEMENTATION

A. *Our approach*

SIRTOS[1]is aiming to be an open source, educational tool. Most of the code required by the operating system, including libraries, was written from scratch. The reason for this decision, is to encourage students to learn how different OS mechanisms work by implementing most of them in a more simple and straight-forward way. Most of the code is written in C and a small part of it is written in Assembly. To make a bootable kernel three things are needed: a code editor, a compiler and a tool to create a bootable image file of the kernel along with the boot-loader. It is very important that students can download the code for free and experiment, but also to be able to build custom versions of it without any cost or proprietary tools. As most Linux distributions freely available, no special software is required to build the code.

B. *Environment*

SIRTOS can be built on any Linux operating system or Windows with a cross-compiler installed. Compiling the code in a Linux distribution will be easier since most of the tools needed are already included in the system.

C. *Compiling*

The code was compiled with the GCC compiler. A makefile script was used to compile each code file and link them together in one binary output file, the kernel. GCC (GNU Compiler Collection) offers a tool-chain for cross compiler set up, which can be installed in both Linux and Windows operating systems.

[1]From Wikipedia, the free encyclopedia: Sirtos (Greek: Συρτός from the Greek: σύρω, siro, "drag [the dance]"), is the collective name of a group of Greek folk dances

D. *Integration to boot-loader*

In order to successfully boot the operating system, we need to integrate it to the boot-loader. The Grand Unified Boot Loader (GRUB) is one of the most common boot-loaders used by many operating systems including most Linux distributions. By using the virtual image disk driver software a bootable image file was created, that can be loaded by either virtual machines or any common computer. There are a many free tools which create a bootable USB drive with the kernel.

IV. DEVELOPMENT

A. *Interrupts*

Interrupts handling is a crucial element in every real time OS and consequently in SIRTOS. Most real time tasks are generated from interrupts such as sensor or counter trigger, clock etc. Therefore, interrupts should have the highest priority to ensure that every interrupt will trigger its handler in time.

To ensure system stability, the interrupt handling mechanism should be conflict-free from tasks accessing and changing the same data from the kernel. For this reason, before any task makes any changes to kernel structures, it first disables interrupts and enables them again when it’s done, creating critical sections.

Another important aspect of the interrupt handling mechanism is the nested interrupts. In some cases, that can lead to two interrupts changing the same data from the kernel. To mitigate this issue interrupts are disabled when an interrupt service routine (ISR) is triggered and re-enabled on return. Thus the code inside an ISR must be as short as possible to avoid missing interrupts.

The goal was to keep the code of the ISRs as simple as possible in order to be easy to understand and also allow for further adjustments, for example a watchdog timer.

B. *Memory management*

SIRTOS is targeted for personal computers and not for embedded systems, therefore is designed with large memory requirements, which are required for personal computers that will need to serve multiple graphical applications and many input and output devices. As a result, SIRTOS supports dynamic memory allocation and paging. Most RTOSs avoid using dynamic memory allocation. On the other hand, dynamic memory allocation can lead to memory fragmentation, if the memory is divided into sections where all of them are small and possibly cannot allocate large continuous blocks of memory. Many real-time operating systems use static memory management but since SIRTOS is targeted for educational purposes and the code of this operating system is less than 15000 lines long, it can easily support dynamic memory allocation without any upcoming issues. Furthermore, including such a mechanism will be an interesting aspect of this project since almost all operating systems use dynamic memory allocation hence it will be interesting for students to understand how it works.Of course it can be also easily excluded from the kernelif other approaches are preferred.

Dynamic memory allocation employs memory swapping, overlays, multiprogramming with a fixed number of tasks, multiprogramming with a variable number of tasks and demand paging [9].

C. *Multitasking*

In real-time operating systems, each task can be in these three states: waiting, ready and executing. Some processes may be waiting for data or other events. Other processes may not be blocked by external events but are not currently executing, these processes are called ready. The transfer of execution from one process to another is called context switch. A context switch needs the state of the running process as well as of the next process to obtain the CPU and to set the CPU state accordingly.

Multitasking is based on the concept of time division, multiplexing and quick task switching creating an illusion to the user, that one processor is serving multiple tasks at the simultaneously. In SIRTOS the tasks are prioritized based on important deadlines that have to be met. For this reason SIRTOS is based on the concept of time periods. Each period lasts a small amount of ticks created by the Programmable Interval Timer (PIT) and has a unique identification number.

Each process has a certain amount to run per period that is given to the task by the operating system which is based on the amount of ready tasks in queue. The formula to assign time to each task to run is the following:

$$T_p = \frac{T_c}{RT} \geq T\min \quad (4)$$

- Where T_p is periodic time to run (ticks),
- T_c is period time (ticks),
- And RT is ready tasks waiting to be executed.
- T_{min} is the minimum running time a task can get to run. This exists to ensure that a task will be given a minimum amount of ticks that is enough to make changes before the task-switch.

In every period the task to run next is the one which has the earliest deadline, hence the highest priority. That task will run for a certain amount of time, and then the next task will be chosen. Every time a task is created, it saves the period number. The period number is also saved before the task-switch. If a task get time to run in every period (periodic time), it will not run again on the same period unless no other tasks are waiting on the queue of ready tasks. This ensures that if a higher priority task starts executing, the illusion of multitasking will not be completely removed resulting in complete system paralysis, but the task will be given the higher priority and the system will ensure that this task will definitely run in every task the amount of time given to this task, while still serving lower priority tasks.

Multitasking and task switching methods are particularly hard to implement. Even the simplest types of multitasking in an operating system require advanced programming skills. The implementation of the scheduling algorithm was the most challenging part of the SIRTOS project, hence there is plenty of room for improvements.

D. Scheduler

The scheduler will be called multiple times in every period with every task switch, and therefore the scheduler's execution time must be reduced for performance reasons. In order to achieve this time reduction, the amount of operations the scheduler will have to perform to get the next, highest-priority task to run has to be minimized.

Furthermore, every time a task is created it needs to be added to the ready task queue. Given that the queue consists of multiple tasks, the scheduler will require too much time to traverse the queue in order to find the task with the highest priority. Thus, it is important to place each task in the correct order inside the ready queue at its time of creation. That is necessary because tasks can be created only once every period but most likely not every period.

As was mentioned in the previous section, every task will be executed only once every period, allowing also lower priority tasks to run. Fig 1 depicts that interrupts triggering ISRs are difficult to handle as the sequence of tasks after an ISR could become unpredictable. Consider the following example: task1, task2 and task3 have a deadline in 3 periods. Since there are three tasks, each will get 1/3 of a period to run and that is enough for all of them to meet their deadline. If task3 gets blocked by an ISR during the second period and task3 completes its second run inside the third period, it won't be able to run again on the third period until all other tasks do, causing it to miss its deadline. As a result, the system will not be able to meet all deadlines.

In most cases, interrupts will trigger the most important and higher priority tasks; hence they are expecting to interrupt all other tasks as we can see in Fig 1. In another scenario a task could be started by an ISR after that ISR completes its execution. If it is a high priority task with a very strict deadline, after that ISR finishes it will enable interrupts and therefore that task could be blocked by another ISR at any time. These tasks should have the ability to block interrupts. This can be implemented in the same way as with critical sections or ISRs, simply by disabling interrupts for a short amount of time.

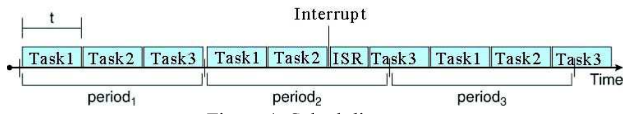

Figure 1. Scheduling

The scheduler is one of the most important parts of every operating system. The main target was to be as simple as possible in order to avoid the issues of software complexity. As there are different types of real time scheduling algorithms and SIRTOS designed to allow for as many as possible to be implemented since it is an educational project.

E. Graphics

It is very important that SIRTOS will be able to support high quality graphics so as to provide an appealing visual experience as a platform of an efficient working environment. Most of the code is written in C, which can be used as low level language similar to Assembly but also framework to create graphical user interface. Creating a GUI allows multiple applications to be developed and tested within the OS. Higher VGA modes require either a graphic card's driver for each type of graphic card the system will use, or a VESA Bios Extension driver (VBE) since almost all systems support VBE. VESA functions usually run in real mode. This means that in order to set a higher resolution mode that VBE can offer, the system will have to switch from protected to real mode and back again. Before switching to real mode, paging will have to be disabled. Modes that support linear address to access the video memory will only work if paging forces this linear address to point to the same physical address so as to access the video memory transparently as if paging was not enabled.

In addition to that, a double buffer is needed in order to refresh the screen. The second buffer must be at the size of the video memory. Every time the refresh function begins, the second buffer is copied into the video memory and thereafter everything is redrawn. The redraw process is important as it depicts changes in the screen. Things that needed to be redrawn can be anything including graphics from desktop, windows, buttons etc. to the mouse pointer.

The GUI can be easily managed with multiple methods. The goal was to keep the graphical system simple and organized and allow for all applications to have a graphic environment which makes it interesting to write small applications and test them within the OS.

F. Input /output Devices

Any personal computer will need to support human interface devices (HID). These are devices that interact directly with humans, by taking input or deliver output. Most common HID devices are the keyboard and mouse. Input-Output devices (I/O) can be both simple devices that don't require high priority and critical devices that require hard real-time deadlines. HID devices don't fall in this category, thus static priorities were assigned to them during the design time.

Each device needs an appropriate driver. Drivers communicate with the operating system through interrupts. SIRTOS does not support nested interrupts ,therefore if too many non-critical devices trigger interrupts it is possible to miss few critical, high priority interrupts. To identify the missed interrupts by the system is too complicated and hard to implement, thus it is important to create mechanisms acting as watchdog timers. The proposed watchdog timers allow ISRs that don't belong to critical devices to run for limited time during periods, where high priority tasks are waiting for critical events. Such an event could be a countdown timer waiting to trigger an actuator to close or open a valve for a water pipe. Again, as an educational operating system, SIRTOS has not been designed with the strictest real time requirements.

G. File system

Every operating system should provide support for at least one file system to store files needed by the user. A virtual file system is almost inevitable before any actual file system can be implemented. During booting, possible before I/O drivers are loaded or even paging enabled code, the Random Access Memory disk is initialized, usually referred to as Initial RAM Disk (InitRD). The InitRD concept is not hard to be implemented and most systems use their own versions of InitRD to store and load important drivers required by the operating system.

The InitRD is loaded by the kernel and stored in the root folder along with the kernel. The InitRD is running into memory and must not be overwritten when paging is initialized. That means that the part of the memory of the InitRD will not be writable but only readable.

H. System requirements

An operating system will need some common system services to provide a friendly user experience. Some of these services include:

Real Time Clock (RTC) is used by personal computers as well as embedded systems and different electronic devices to keep track of time and date.

PS/2-USB keyboard and mouse drivers are straight forward to build but essential to implement as they are the primary user input devices and they will be used almost in the same way for both PS/2and USB devices. Most Basic Input Output Systems (BIOS) use the USB legacy support option by default, since this option allows even USB devices such as the keyboard and mouse to be used like common PS/2 devices.

Most operating systems have their own boot loader. The installation of an operating system requires a mechanism of loading the operating system. Supporting multiple operating systems in one machine can be difficult, but in most cases is an essential requirement. Therefore if a boot loader doesn't perform exactly as expected then the operating system will not load successfully. To address this problem SIRTOS supports multi-boot specification. Multi-boot specification describes the interface between a boot loader and an operating system. Thus, every boot loader that follows this multi-boot specification will be able to load all operating systems that comply with this specification. As a result by adding the latest multi-boot specification, SIRTOS can be booted using the Grand Unified Boot Loader (GRUB) along with other Linux based operating systems that also support GRUB.

I. Security

Building a reliable system requires a multi-layered security approach to prevent unauthorized data access, as well as, system failure. Real-time operating systems operate under very demanding conditions.

Creating protection rings one can addresses one of these challenges. Each protection ring provides a different access limit as shown in Fig 2.

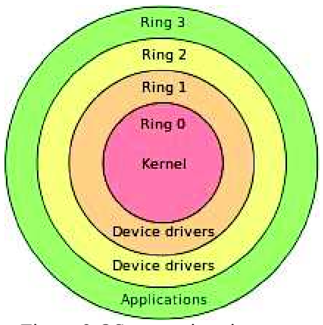

Figure 2.OS protection rings

The best approach to ensure that protection rings will not be breached is by having a multi-user system where each user has different privileges. By giving different privileges to users, every user has limited access and can only enter specific protection rings according to these privileges. Hence user identification is a crucial component of SIRTOS.

User identification is performed through common login procedures where a hashing algorithm, such as the MD5 algorithm, converts passwords to hash values. An MD5 algorithm of 128bit (16 bytes) can safely be used to store passwords. Each time that a password is entered, the algorithm converts the password into a hash value, which is compared to the stored value for the specific user. If the values match the identification is considered successful.

Security in operating systems is a nonetheless critical issue and requires the implementation of multiple mechanisms and techniques. The implementation of basic encryption algorithms is a basic step towards a more secure and reliable operating system.

J. Related Works

One operating system created for educational purposes is MINIX, short for Minimal UNIX. MINIX is a close cousin of GNU/Linux. It was originally developed in 1987 by Andrew S. Tanenbaum as a teaching tool for his textbook Operating Systems Design and Implementation. Today MINIX is an open source operating system with a kernel of less than 12.000 lines of code. MINIX's hardware requirements are not be a major concern. Requiring 16MB of RAM and a gigabyte of hard drive space, MINIX should install on most computers made in the last decade [11].

V. CONCLUDING REMARKS AND FUTURE WORK

During this research many problems have arisen and addressed. One of the most difficult to tackle is the problem of many high priority real-time processes running on the system along with normal non-real-time processes. SIRTOS goal was to be an operating system for personal computers that will be able to handle real-time processes, almost as well as any commercial real time system but also keep the multitasking performance as high as can be for improved user experience. Solving this issue was crucial in moving forward with this project. The way that this issue has been resolved was by giving the scheduler a dynamic priority-driven algorithm, similar to an EDF, in which a task's priority is given according to its deadline. The scheduling algorithm takes also into account the periodic running time dynamically calculated by the scheduler according to system load. After numerous experiments and tests, it has been observed that merging an EDF algorithm with some principles from a RM algorithm resulted in a better multitasking experience and higher tasking performance, while still maintaining ability to meet deadlines.

By combining all these together, SIRTOS has been built as an open source, simple, real-time operating system from scratch. The code is less than 15000 lines long, which most of them are written in C while a few them in assembly. SIRTOS offers basic applications such as a paint program, a control panel, a calculator, and different settings to test graphics, a terminal for basic commands, a system panel and a memory manager panel to test how memory is allocated. In addition to that different user office tools are available such as a text editor and a file manager. The system requires only 12 megabytes of RAM to run. SIRTOS can handle multiple tasks without degrading its performance. SIRTOS can be practically used as a virtual machine or booted in a personal computer. There are various features to implement beyond this point. In particular, many different services and applications can be added, as for example the ability for the operating system to load and execute both ELF and binary executable files. Nonetheless the important of SIRTOS is that anyone can write software and extend the abilities of this operating system. It is a unique open source tool for students and professional programmers to study and explore operating system development and as well its basic aspects. A visual representation of SIRTOS can be seen in Fig 3.

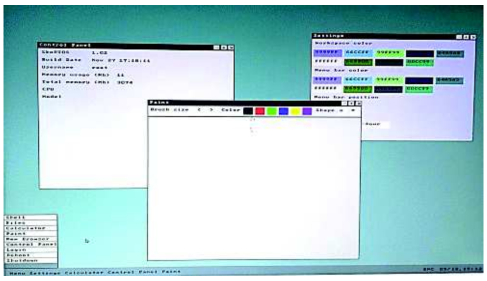

Figure 3. SIRTOS

ACKNOWLEDGMENT

The authors would like to kindly acknowledge the OS-Dev wiki network for providing large amount of information and code on operating system development.

REFERENCES

[1] James H. Anderson, Srikanth Ramamurthy, Kevin Je ay, "Real-Time Computing with Lock-Free Shared Objects",IEEE Real-Time Systems Symposium, 1995.

[2] Tarek F. Abdelzaher , Vivek Sharma , Chenyang Lu, "A Utilization Bound for Aperiodic Tasks and Priority Driven Scheduling", IEEE Transactions on Computers, 2004.

[3] David Steward and Michael Barr "Introduction to Rate Monotonic Scheduling", Netrino, November 2007.

[4] Yanbing Li, Miodrag Potkonjak, Wayne Wolf, "Real-Time Operating Systems for Embedded Computing", Department of Eletrical Engineering, Prinston University Department of Computer Science, 1997.

[5] S. Baruah, "Fairness in periodic real-time scheduling", Proceedings of the 16th IEEE Real-time Systems Symposium, pp. 200–209, 1995.

[6] Jagbeer Singh, Bichitrananda Patra, Satyendra Prasad Singh "An Algorithm to Reduce the Time Complexity ofEarliest Deadline First Scheduling Algorithm in Real-Time System", (IJACSA) International Journal of Advanced Computer Science and Applications,Vol. 2, No.2, February 2011.

[7] S. Lauzac, R. Melhem, and D. Mosses,"Compression of Global andPartitioning Scheme for Scheduling Rate Monotonic Task on a Multiprocessor", The 10th EUROMICRO Workshop on Real-Time Systems, Berlin,pp.188-195, 1998.

[8] Mohamed Marouf , Inria Rocquencourt , Yves Sorel. "Scheduling non-preemptive hard real-time tasks with strict periods", Proceedings of 16th IEEE International Conference on Emerging Technologies and Factory Automation ETFA, 2011.

[9] S. Baskiyar, and N. Meghanathan "A Survey of Contemporary Real-timeOperating Systems" Informatica, 2005.

[10] G. C. Buttazzo, "Hard Real-Time Computing Systems: predictable scheduling algorithms and applications," Springer company, 2005.

[11] Andrew S. Tanenbaum, "Operating Systems Design and Implementation, Third Edition",Vrije Universiteit Amsterdam, The Netherlands, 2006.

[12] Mehrin Rouhifar , Reza Ravanmehr, "A Survey on Scheduling Approaches for Hard Real-Time Systems", International Journal of Computer Applications, 2015.

[13] C.M. Krishna and Shin K.G. Real-Time Systems. Tata McGrawiHill, 1997.