REAL-TIME OPERATING SYSTEM

Lokesh Madan[1], Kislay Anand[2] and Bharat Bhushan[3]

[1]Department of Computer Science, Dronacharya College of Engineering, Gurgaon, India

[2]Department of Computer Science, Dronacharya College of Engineering, Gurgaon, India

[3]Department of Computer Science, Dronacharya College of Engineering, Gurgaon, India

ABSTRACT

Real-time systems play a considerable role in our society, and they cover a spectrum from the very simple to the very complex. Examples of current real-time systems include the control of domestic appliances like washing machines and televisions, the control of automobile engines, telecommunication switching systems, military command and control systems, industrial process control, flight control systems, and space shuttle and aircraft avionics.

All of these involve gathering data from the environment, processing of gathered data, and providing timely response. A concept of time is the distinguishing issue between real-time and non-real-time systems. When a usual design goal for non- real-time systems is to maximize system's throughput, the goal for real-time system design is to guarantee, that all tasks are processed within a given time. The taxonomy of time introduces special aspects for real-time system research. Real-time operating systems are an integral part of real-time systems. Future systems will be much larger, more widely distributed, and will be expected to perform a constantly changing set of duties in dynamic environments. This also sets more requirements for future real-time operating systems.

This seminar has the humble aim to convey the main ideas on Real Time System and Real Time Operating System design and implementation.

INTRODUCTION

Timeliness is the single most important aspect of a real -time system. These systems respond to a series of external inputs, which arrive in an unpredictable fashion. The real-time systems process these inputs, take appropriate decisions and also generate output necessary to control the peripherals connected to them. As defined by Donald Gillies "A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time in which the result is produced. If the timing constraints are not met, system failure is said to have occurred."

It is essential that the timing constraints of the system are guaranteed to be met.

Guaranteeing timing behaviour requires that the system be predictable.

The design of a real -time system must specify the timing requirements of the system and ensure that the system performance is both correct and timely. There are three types of time constraints:

- Hard: A late response is incorrect and implies a system failure. An example of such a system is of medical equipment monitoring vital functions of a human body, where a late response would be considered as a failure.
- Soft: Timeliness requirements are defined by using an average response time. If a single computation is late, it is not usually significant, although repeated late computation can result in system failures. An example of such a system includes airlines reservation systems.
- Firm: This is a combination of both hard and soft timeliness requirements. The computation has a shorter soft requirement and a longer hard requirement. For example, a patient ventilator must mechanically ventilate the patient a certain amount in a given time period. A few seconds' delay in the initiation of breath is allowed, but not more than that.

OVERVIEW

The following table compares some of the key features of real -time software systems with other conventional software systems.

Feature	Sequential Programming	Concurrent Programming	Real Time Programming
Execution	Predetermined order	Multiple sequential programs executing in parallel	Usually composed of concurrent programs
Numeric Results	Independent of program execution speed	Generally dependent on program execution speed	Dependent on program execution speed
Examples	Accounting,	UNIX operating	Air flight

REAL-TIME PROGRAMS: THE COMPUTATIONAL MODEL

A simple real -time program can be defined as a program P that receives an event from a sensor every T units of time and in the worst case, an event requires C units of computation time. Assume that the processing of each event must always be completed before the arrival of the next event (i.e., when there is no buffering). Let the deadline for completing the computation be D. If D < C, the deadline cannot be met. If T < D, the program must still process each event in a time O/ T, if no events are to be lost. Thus the deadline is effectively bounded by T and we need to handle those cases where C O/ D O/T.

DESIGN ISSUE OF REAL TIME SYSTEMS

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. A common misconception is to consider, that real-time computing is equivalent to fast computing. In traditional non-real- time computer systems, the performance goal is throughput: as many tasks should be processed as possible in given time period. Real-time systems have a different goal to meet: as many tasks as possible should be executed so, that they will complete and produce results before their time limit expires. In other words, the behavior of real-time system must be predictable in all situations.
To achieve predictability, all components of the real-time system must be time bounded. A predictability of the system depends on many different aspects.

 The computer hardware must not introduce unpredictable delays into program execution. For example, caching and swapping as well as DMA cycle stealing are often problematic when determining process execution timing.
 An operating system must have a predictable behavior in all situations. Often the common-purpose operating systems, like UNIX, are too large and complex, and they have too much unpredictability. Thus, a special microkernel operating systems like the Chorus microkernel have been designed for real-time purposes. Also traditional programming concepts and languages are often not good for real- time programming. No language construct should take arbitrary long to execute, and all synchronization, communication, or device accessing should be expressible through time-bounded constructs. However, despite all these real-time requirements could be solved, a human factor - the real-time programmer can always cause unpredictability to the system. To assist the programming process, numerous methods have been produced for real-time system design, specification, verification, and debugging.
Typically, a real-time system consists of controlling system and a controlled system. The controlled system can be viewed as the environment with which the computer interacts. The typical real-time system gather information from the various sensors, process information and produce results. The environment for real-time system may be highly in deterministic. Events may have

unpredictable starting time, duration and frequency. However, real-time system must react to all of these events within prespecific time and produce adequate reaction.

To guarantee, that a real-time system has always a correct view of its environment, a consistency must be maintained between them. The consistency is time-based. The controlling system must scan its environment fast enough to keep track changes in the system. The adequate fastness depends on application. For example, sensing a temperature needs often slower tempo than sensing a moving object.

The need to maintain consistency between the environment and the controlling system leads to the notion of temporal consistency. Temporal consistency has two components:

1. Absolute consistency between the state of the environment and the controlling system. The controlling system's view from the environment must be temporally consistent, it must not be too old.

2. Relative consistency among the data used to derive other data. Sometimes, the data items depend on each other. If a real-time system uses all dependent values, they must be temporally consistent to each other.

The deadlines have been divided into three types: hard, soft and firm deadlines. The hard deadline means, that a task may cause very high negative value to the system, if the computation is not completed before deadline.

Opposite to this, in a soft deadline, a computation has the decreasing value after the deadline, and the value may become zero at later time. In the middle of these extremes, a firm deadline is defined: a task loses its value after deadline, but no negative consequence will occur. A real-time system must guarantee, that all hard deadlines and as many as possible of other deadlines are met.

SCHEDULING

To support timeliness in a real-time system, a special real-time task scheduling methods have been introduced. Traditional real-time research has been concerned to uni processor cheduling. As a complexity and scale of real-time system grows, the processing power of real-time system is increased by adding new processors or by distribution. These issues introduce several new concepts to scheduling research, numerous schemes have been introduced for multiprocessor and distributed scheduling. In this section, we discuss two main principles for real- time scheduling, a scheduling paradigm and a priority inversion problem.

SCHEDULING PARADIGMS

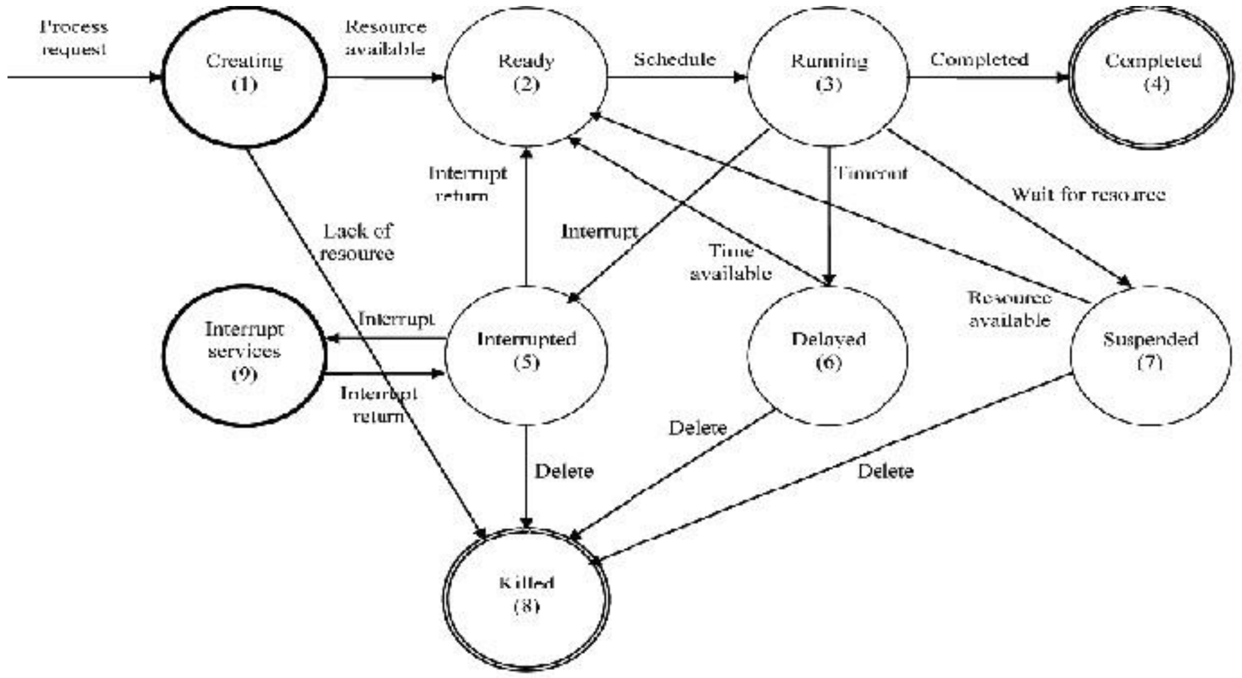
The simplest real-time scheduling method is not to schedule at all. This method sounds dummy, but in many real-time systems, only one task exists. A common example is a typical programmable logic. Programmable logics are widely used in the industry automation. A programmable logic's program is executed periodically. During every execution, the program reads all inputs, makes simple calculations and sets appropriate outputs. Same program is executed at every time, and a state of the system depends on internal variables set in the previous runs. However, interrupts can be used to catch asynchronous events, but an advanced processing must be made within standard run periods.

In more advanced real-time systems, several simultaneous tasks can be executed. Every task may have different timing constraints and they may have a periodic or an aperiodic execution behavior. The periodic behavior mens, that a task must be executed within prespecific time intervals. When a task has an aperiodic behavior, it will execute, when an external stimulus occurs. In a typical real-time systems, both type of tasks exists. However, all aperiodic tasks can always be transformed to periodic. A task structure of the system describes, when the tasks can be started. Many systems have a static task structure, where tasks are installed during system startup and no new tasks can be started afterwards. In dynamic task structure, tasks can be started and ended during system uptime. Depending on particular system's behavior, the different scheduling paradigms have been introduced:

1. Static table-driven approaches: These perform static schedulability analysis and the resulting schedule (table) is used at run time to decide, when a task must begin execution. This is a highly predictable approach, but it is very inflexible, because a table must always be reconstructed, when a new task is added to the system. Due to predictability, this is often used when absolute hard deadlines are needed.

2. Static priority-driven pre-emptive approaches: These perform static schedulability analysis, but unlike in the previous approach, no explicit schedule is constructed. At run time, tasks are executed "highest priority first". This is a quite commonly used approach in concrete real-time systems.

3. Dynamic planning-based approaches: Unlike the previous two approaches, feasibility is checked at run time. A dynamically arriving task is accepted for execution only if it is found feasible.



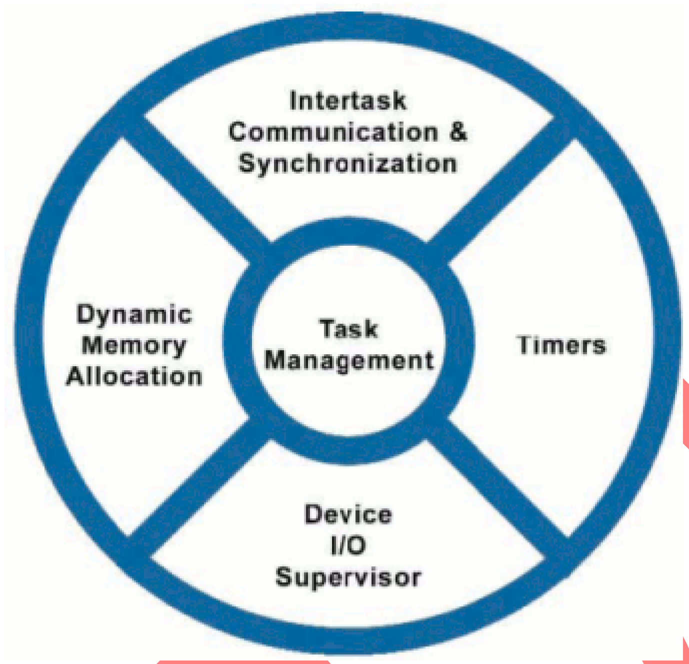
4. Dynamic best effort approaches: Here no feasibility checking is done. The system tries to do its best to meet deadlines, but a task may be aborted during its execution.

Unfortunately the most scheduling problems are NP-complete. However, many good heuristic approaches have been presented. Thus, numerous algorithms have been introduced to support these scheduling paradigms. Algorithms are either based on the single scheduling paradigm or they can spread over several paradigms. These algorithms include least common multiply (LCM) method, earliest deadline first (EDF) method, rate monotonic (RM), and many others. A survey of scheduling methods is found in.

REAL-TIME OPERATING SYSTEM

Real-time operating systems are an integral part of real-time systems. Examples of these systems are process control systems, flight control systems and all other systems that require result of computation to be produced in certain time schedule. Nowadays real-time computing systems are applied to more dynamic and complex missions, and their timing constraints and characteristics are becoming more difficult to manage. Early systems consisted of a small number of processors performing statically determined set of duties. Future systems will be much larger, more widely distributed, and will be expected to perform a constantly changing set of duties in dynamic environments. This also sets more requirements for future real-time operating systems.

Real-time operating systems need to be significantly different from those in traditional time sharing system architectures because they have to be able to handle the added complexity of time constraints, flexible operations, predictability and dependability.


Intertask Communication & Synchronization
Dynamic Memory Allocation
Task Management
Timers
Device I/O Supervisor

REAL-TIME OPERATING SYSTEM REQUIREMENTS AND BASIC ABSTRACTIONS

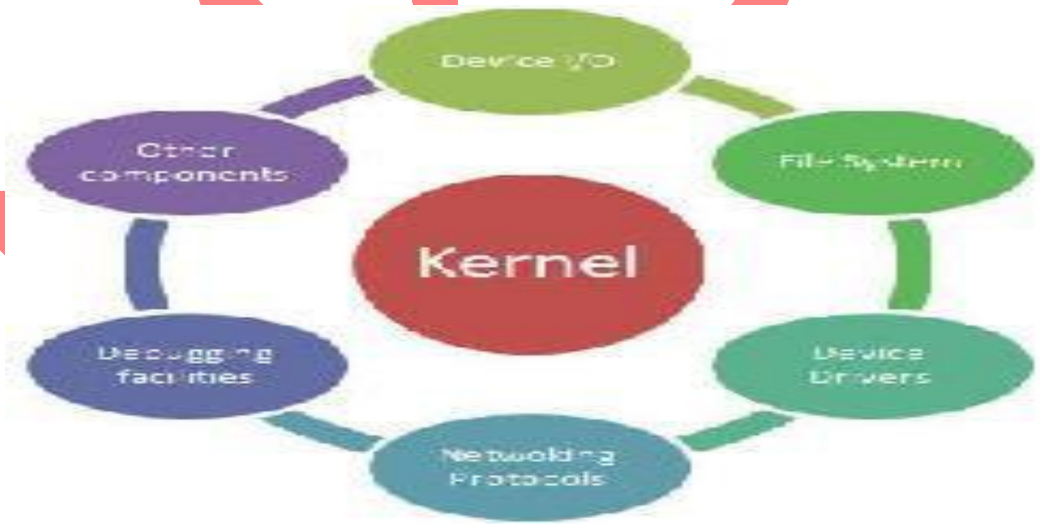
In real-time systems, operating system plays a considerable role. The most important task of it is to schedule system execution (processes) and make sure that all requirements that the system is meeting are filled. In this chapter we will introduce some general terms used in operating systems and real-time systems:

GENERAL TERMS

Real-time operating systems require certain tasks to be computed within strict time constraints. Time constraints define deadline, the response time in which the computation has to be completed. There are three different kinds of deadlines; hard, soft and firm. Hard deadline means that if the computation is not completed before deadline, it may cause a total system failure. Soft deadline means that computing has a decreasing value after the deadline but not a zero or negative number. If firm deadline is missed, the value of task is lost but no negative consequence is occurred.

Fault tolerance, the capability of a computer, subsystem or program to withstand the effects of internal faults, is also a common requirement for real-time operating systems.
Several real-time systems collect data from their environment at predetermined time intervals which requires periodic execution.

 Predictability
One common denominator in real-time systems seems to be that all designers want their real-time systems to be predictable. Predictability means, that it should be possible to show, demonstrate, or prove that requirements are met subject to any assumptions made, for example, concerning failures and workloads. In other words predictability is always subject to the underlying assumptions made.
For static real-time systems in deterministic, stable environments we can easily predict the overall system performance over large time frames as well as predict the performance of some individual task. In more complicated, changing, nondeterministic environment, for example a future system of robots co- operating on Mars, it is far more complicated task to predict in design phase, how the system is actually going to act in its real environment. In operating system level system must be designed to be so simple, that it is possible to predict worst- case situations in execution.



In addition to the POSIX, other basic requirements described on a recent survey of
RTOS are the following:

- Low overhead: The context-switch times for threads and the OS overhead should be minimal.
- Preemptive: The RTOS must be able to preempt the currently executing thread to give the CPU to a higher-priority thread. Again, this feature is

needed to allow an urgent event (e.g., safe temperature exceeded) to preempt an activity of lower criticality.

- Deterministic synchronization: Ability for multiple threads to communicate among themselves within a predictable time.
- Priority levels: The RTOS must provide sufficient priority levels to allow for effective application implementation. This feature is important in enabling flexible preemptive priority scheduling.
- Predefined latencies: The timing of API calls must provide expected latencies. This feature is helpful when process steps, such as adding, mixing, or heating, need to be commenced within a certain amount of time of a preceding process step.

The features of an RTOS are necessary, but not sufficient, to implement a real- time system. Whether or not an RTOS provides the necessary features for your system is worthless if the underlying hardware does not provide the necessary horsepower. The CPU speed, the memory access speed, and the device access speed define the potential speed of the underlying hardware. However, industrial and laboratory automation software can present a substantial processing load. Therefore, you need to ensure that the hardware is capable of supporting the hard real-time constraints and worst-case scenario in your fully loaded system regardless of the underlying RTOS. Once the hardware is proven to be suitable for your real-time application, then you can evaluate the features of RTOSs.

EVALUATING AN RTOS

Whether you write your own software for lab automation, write middleware to help communicate between programs, or use off-the-shelf software, it is beneficial to know how to select the right RTOS for your environment. Of course, the first question you should always consider is if an off-the-shelf RTOS will be compatible with the automation software that you need to host. Many commercial RTOSs and automation software developers have such compatibility lists.
A real-time application for an embedded system will often require an RTOS with a very small footprint and little overhead. A real-time application running on a PC or Mac can support more complex RTOSs that provide additional features and sophisticated user interfaces. Both hardware and software choices should be considered together when designing a real-time system. The system cost, the development time, and the chance of success will depend on the choices you made. One of the most important criteria, often overlooked, when evaluating a RTOS is the availability of development tools such as real-time symbolic debuggers and appropriate programming language support. The use of sophisticated IDE is common today. IDEs provide developers with tools and examples to quickly develop your software, test it, and manage different revisions.

There are many non-technical criteria that must be considered as well. Your decision will dictate the cost of ownership for your company. For example, you may choose a commercial RTOS or one from the open source community. In certain extreme cases (such as for low-production volume high-reliability systems), you may even need to develop an RTOS from scratch. For any of these cases, you must evaluate what is going to be the total cost to the company. Commercial RTOS products may have a high up-front cost and may charge royalty fees for every deployment of your system. You may also consider obtaining the source code, which will add to the cost. The open source RTOS might offer free access to source and binaries with no royalties, but often require the purchase of support and long-term commitment from the user community. Open source RTOS may lead to higher long-term costs due to reliance on low-level development tools and highly technical individuals. In summary, the total cost of ownership depends on both up-front and long-term costs of royalties, support, maintainability, training, and consulting services.
There are also many technical decision criteria to be considered. In any case, the implementation of the POSIX features provided in Characteristics of RTOSs would be a good place to start, because the first thing you need to do is to choose between a thread-or process-based RTOS

A SURVEY OF RTOSS

There are over 30 RTOSs, which can be classified as open source versus commercial. Here we provide a summary of the features of a few of the top 10 RTOSs listed in a 2005 survey by Embedded System Design.
The VxWorks commercial RTOS from Wind River is the most widely adopted in the embedded industry (e.g., it is used on the International Space Station). VxWorks was first released in the early 1980s and provides a flexible API with more than 1800 methods. The development host can be Red Hat Linux, Solaris, SuSE Linux, Windows 2000 Professional, or Windows XP. VxWorks is available for all popular CPU platforms: x86, PowerPC, ARM, MIPS, 68K, CPU 32, ColdFire, MCORE, Pentium, i960, SH, SPARC, NEC V8xx, M32 R/D, RAD6000, ST 20, and TriCore. The kernel supports preemptive priority scheduling with 256 priority levels and round- robin scheduling. VxWorks is a multithreading RTOS that provides deterministic context switching and supports semaphores and mutual exclusion with inheritance.
A unique feature of Windows CE is the concept of fibers. A fiber is a unit of execution that must be manually scheduled by the application. A fiber is an execution unit that runs within the context of the thread that schedules it. A thread can schedule multiple fibers but they are not preemptively scheduled. The thread schedules a fiber by switching to it from another fiber. The running fiber assumes the identity of the thread. Fibers are useful in situations where the application needs to schedule its own threads.
The most widely adopted free, open source RTOS, eCos (embedded Configurable operating system) was released in 1986. eCos provides a graphical-configuration tool and a command line-configuration tool to customize and adapt the RTOS to meet application-specific requirements. This feature allows the user to set the OS to the desired memory footprint and performance requirements. Development hosts

are Windows and Linux and the supported target processors are x86, PowerPC, ARM, MIPS, Altera NIOS II, Calmrisc16/32, Freescale 68k ColdFire, Fujitsu FR-V, Hitachi H8, Hitachi SuperH, Matsushita AM3x, NEC V850, and SPARC. The eCos kernel can be configured with the bitmap scheduler or the multilevel queue (MLQ) scheduler. Both schedulers support priority-based scheduling with up to 32 priority levels. The bitmap scheduler is somewhat more efficient and only allows one thread per priority level. The MLQ scheduler allows multiple threads to run at the same priority. First in, first out (FIFO) or round-robin is used to schedule threads with the same priority. The eCos RTOS supports OS primitives such as mutexes, semaphores, mailboxes, and events for synchronization and communication between threads.

Contemporary OSs such as Linux and Windows XP, called XP Embedded, also have extensions that enable them to support real-time applications. But these OS are only suitable for large real-time systems due to footprint required. On the other hand, there is no need to use specialized tools and there are a large number of developers who can quickly learn how to make use of the real-time features.

CONCLUSION

This paper gives an overview of the real-time system issues. The main concept in all real time systems is predictability. The predictability depends on numerous different aspects, hardware, operating system, programming languages, and program design and implementation methods. The real-time system must also have a temporally consistent view of the environment it belongs to. The consistency can be addressed as terms of absolute and relative consistency. The tasks in real-time systems may have arbitrary deadlines. The deadlines have been divided into three categories, depending on how the system is affected if a deadline is missed. These categories are hard, firm and soft deadlines.

To support timeliness and predictability, the different scheduling methods have been produced for uniprocessor systems as well as multiprocessor and distributed systems. Tasks in real-time systems can have periodic or aperiodic behavior, and several scheduling paradigms have been produced to support different system types. When resources need to be allocated in real-time systems, a priority inversion problem may arise. The priority inversion may cause the task to lose its deadline. Several priority inheritance protocols have been introduced to solve the priority inversion problem.

Real-time operating system is an integral part of real-time system. Real-time operating system offers tools to guarantee predictability. The small, proprietary kernels have been stripped out and optimized for real-time purposes. These kernels may be scalable from simple embedded systems to offering full POSIX/UNIX support. Also commercial timesharing operating systems can be extended to meet the real-time requirements, but these are not often applicable for real-time performance requirements. Also several research kernels were introduced. These kernels are used to develop and demonstrate new real-time system features.

Real-time database is an example case of real-time system. It combines database and realtime concepts. These concepts include handling a database schema, efficient data management support, transactionality, failure recovery mechanisms, data temporality, and real-time access to data.

REFERENCES

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions. ACM SIGMOD Record.

[2] P. Elovaara and K. Raatikainen. Evaluation of concurrency control algorithms for realtime databases. Report C-1996-52, University of Helsinki, Dept. of Computer Science, Helsinki, Finland.

[3] B. Furht, D. Grostick, D. Gloch, G. Rabbat, J. Parker, and M. Mc Roberts. Real-Time Unix Systems Design and Application Guide, Kluwer Academic Publishers.

[4] W. Halang. Implication on suitable multiprocessor structures and virtual storage management when applying a feasible scheduling algorithm, in hard real- time environments. Software Practice and Experience.

[5] K. Kavi and S. Yang. Real-time system design methodologies: An introduction and a survey, Journal of Systems and Software.

[6] Mathai Joseph, Real -time Systems: Specification, Verification and Analysis, Prentice Hall International, London.

[7] Bran Selic, Turning Clockwise: Using UML in the Real -Time Domain, Communications of the ACM.

[8] K. Ramamritham and J. Stankovitc. Scheduling algorithms and operating system support for real-time systems.

[9] K. Ramamritham. Real-time databases. Distributed and parallel databases,.

[10] J. Bacon, Concurrent systems.