

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.

1. 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.

One need to distinguish between on-line systems such as an airline reservation system, which operates in real-time but with much less severe timeliness constraints than, say, a missile control system or a telephone switch. An interactive system with better response time is not a real-time system. These types of systems are often referred to as soft real time systems. In a soft real-time system (such as the airline reservation system) late data is still good data. However, for hard real-time systems, late data is bad data. In this paper we concentrate on the hard and firm real-time systems only.

Most real-time systems interface with and control hardware directly. The software for such systems is mostly custom-developed. Real-time Applications can be either embedded applications or non-embedded (desktop) applications. Real-time systems often do not have standard peripherals associated with a desktop computer, namely the keyboard, mouse or conventional display monitors. In most instances, real-time systems have a customized version of these devices.

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, payroll	UNIX operating system	Air flight controller

1.1 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 $C > D > O(T)$.

2. *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 [17]. 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 [16] 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 [4].
- 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 [9] 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 [19]. 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 [5].

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 indeterministic. 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 [12]:

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.

There are several possibilities to maintain temporal consistency. The state of the environment is often scanned in periodical basis and an image of the environment is maintained in the controlling system. A timestamp methodology is often used to figure out validity of the system's image of the environment.

A typical real-time system consists of several tasks, which must be executed in simultaneous manner. Each task has a value which is gained to the system if a computations finishes in specific time. Each task has a deadline, which indicates a time limit, when a result of the computing

becomes useless, ie. gains zero or negative value to the system. Furthermore, a task may have some value after the deadline has been expired.

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. Figure 1 plots the value versus time behavior for different deadline types. A real-time system must guarantee, that all hard deadlines and as many as possible of other deadlines are met.

Examples of deadline types are quite intuitive. A typical example of hard deadline can be found from the system controlling the nuclear power plant: adding coolant to the reactor must be done before temperature gets too high. A typical example for firm deadline can be found from the industry automation. A real-time system attempts to recognize a moving object. An object can be scanned only, when it is in the sight of the scanning device. If the object is not recognized, it can be rejected or rescanned later. Thus, an operation loses its value after a certain time period, but no harm will occur due to failure. A typical example of soft deadline is a combined operation sequence. The whole sequence has a firm deadline, but if some of the components of the sequence miss their deadlines, the overall sequence might still be able to make its deadline.

As a summary, the timing requirements for real-time systems can be expressed as following constraints to the processing [5]:

1. Response time, deadline: the system must respond to the environment within a specified time after input (or stimuli) is recognized.
2. Validity of data: in some cases, the validity of the input or output is a function of time. That is, some stimulus and the corresponding response become obsolete with time, and the time interval for which data is valid must be accounted in processing requirements.

3. Periodic execution: in many control systems, sensors collect data at predetermined time intervals.

4. Coordinating inputs and outputs: in some applications, input data from various sensors must be synchronized. Otherwise, decisions would be made based on inconsistent information.

An example of real-time system is simplified unmanned vehicle system (SUVS) [20]. The SUVS controls a vehicle with no assistance from a human driver. It periodically receives data from sensors such as the speedometer, temperature sensor, and direction sensor. It controls the vehicle by generating appropriate signals to actuators, such as the accelerator, brake and steering wheel.

Decisions are made based on the current inputs from the sensors and the current status of the road and the vehicle. Decisions must be made within a specified time. The events can occur very unexpectedly. Let's think about a scenario, where an obstacle suddenly falls into the road. The braking and steering decisions must be done within a short time period to avoid crashing. These decisions must also be synchronized and the status of the road and other traffic must be taken into account. Thus, most of tasks in SUVS have a hard deadline. This is typical in many safety critical systems [6, 7]. A real-time behavior is often essential when designing a safety critical system.

3. 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 uniprocessor scheduling. 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.

3.1 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 means, 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 [13]:

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 [13].

3.2 Priority inversion problem

In a multitasking environment, the shared resources are often used. The usage of these resources must be protected with a well-known methods, like semaphores and mutexes. However, a priority inversion problem [14] arises, if these methods are used in real-time system with a pre-emptive priority-driven scheduling. In priority-driven scheduling, the eligible task with a highest priority is always executed. The task is eligible, when it is not waiting for any event, so when it is runnable. When the higher priority task becomes eligible, it is always started to run. The task with a lower priority is pre-empted, the task is released from a processor in favor to higher priority task.

A priority inversion problem arises, when a lower priority task has reserved a shared resource. If a higher priority task needs the same resource, the resource cannot be acquired, because it is already reserved by another task. This forces the higher priority task to block, which may lead to the missing of its deadline. Also the deadlock situation is possible. Figure 2 illustrates an execution sequence, where a priority inversion occurs. A task T3 executes and reserves a resource. A higher priority task T1 pre-empts task T3, but wants to allocate a resource reserved by task T3. Then, task T2 becomes eligible and blocks task T3. Because the task T3 cannot be executed, the resource remains reserved suppressing task T1 to run. Thus, the task T1 misses its deadline due to resource conflict.

Several approaches have been introduced to rectify the priority inversion problem. The use of priority inversion protocols is one approach. The basic idea of priority inversion protocols is that when a task blocks one or more higher priority tasks, it ignores its original priority assignment and executes its critical section at the highest priority level of all the tasks it blocks. After exiting its critical section, the task returns its original priority. Figure 3 illustrates, how a priority inversion problem presented in the figure 2 can be solved with a priority inheritance protocol. Again, the task T3 executes and reserves a resource, and a higher priority task T1 wants to allocate that resource. As a result of priority inheritance, the task T3 inherits priority of T1 and executes its critical section in that priority. Thus, the task T2 cannot pre-empt task T3 and the resource is released after a critical

Real-time systems

www.bestneo.com

section is finished. Now the task T1 can acquire resource and it is completed to its deadline.

4. 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.

5. 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:

5.1 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.

5.2 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.[2]

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.

5.3 Temporal consistency

The controlling system must scan its environment fast enough to maintain a correct view of it. If that is taken care of, it can be said that consistency is achieved between real-time system and its environment. This leads to notion of temporal consistency [6], which has two components;

- Absolute consistency, which is achieved between the controlling system and the state of the environment if the systems view of the environment is temporally consistent.
- Relative consistency, which is achieved if data items dependent of each other in the system are temporally consistent to each other.
- Temporal consistency is usually maintained by periodically scanning the environment and maintaining an image of the environment in the controlling system.

6. *Real-time operating system structure*

When systems have become larger and unwieldy, they are difficult to comprehend, develop and maintain.

A recent trend in operating system development consists of structuring the operating system as a modular set of system servers which sit on top of a minimal kernel, microkernel, rather than using the traditional monolithic kernel, in which all functionality of the system is provided. Figure 3.1 presents monolithic kernel of the UNIX system [7].

3.1 Monolithic kernel of the UNIX system

The idea in microkernel-based operating systems are that those functions which are needed universally, by every component of the system, form the microkernel. Other functionality is handled outside the kernel and they can be specially tailored for each application. Notations close and open system are also used for this purpose. Real-time operating system are also usually based on microkernel architecture and must support these main functional areas [1]:

- process management and synchronisation
- memory management
- interprocess communication
- I/O

3.2 Structure of the CHORUS system

Microkernel based operating systems are like stripped down timesharing operating systems with only minimal functionality. In figure 3.2 microkernel of the CHORUS system is presented [7].

In real-time operating systems microkernels are used to achieve strict timing requirements. The structure of system must be compact, all unnecessary functionality must be stripped down and execution time of a single task must be minimized. In real-time operating systems reaction time

has to be short and the scheduling algorithm to be executed must be fast and very simple.

6.1 Basic abstractions

Usually current real-time operating systems support following basic abstractions with slight differences between different systems:

Task : The basic unit of resource allocation. This includes /actor address space and access to the system resources.

Thread : The basic unit of execution. Usually executed in the address space of a single task.

Port : A one-way communication channel implemented as a message queue.

Port set : A group of ports. Port set has usually only one queue.

Message : Collection of data objects used in communicating between threads.

7. Real Time Operating System Types

They can be divided into 3 categories :

- small proprietary kernels
- real-time extensions to commercial timesharing operating systems
- research kernels.

7.1 Small Proprietary Kernels

The small, proprietary kernels are often used for small embedded systems when very fast and highly predictable execution must be guaranteed. The kernels are often stripped down and optimized to reduce the run-time overhead caused by the kernel. The usual characteristics of these kernels are [13]:

- fast context switching.
- small size and stripped-down functionality.
- quick response to external interrupts.
- minimized intervals, when interrupts are disabled.
- fixed or variable sized partitions for memory management.
- ability to lock code and data into memory.
- special sequential files for data accessing.

To deal with timing requirements, the kernel should provide a priority scheduling mechanism and a bounded execution time for most primitives. For timing purposes, the kernel should maintain a real-time clock

and provide special alarm and timeout services, as well as primitives for delay by a fixed amount of time. In general, the kernel also performs multitasking and intertask communication and synchronization via standard well-known constructs such as mailboxes, events, signals, and semaphores. Additionally, the kernel should support real-time queuing methods such as delivering messages by a priority order. These kernels are suitable for small applications, such as instrumentation, communication front ends, intelligent peripherals, and process control. Since the applications are simple, it is relatively easy to determine that all timing constraints are met. As complexity of the system increases, it becomes more and more difficult to map all timing, computation time, resource, and value requirements to a single priority for each task. In these situations demonstrating predictability becomes very difficult. Because of these reasons, some researchers believe that more sophisticated kernels are needed to address timing and fault tolerance constraints. Recently there are efforts to produce scalable kernels. The smallest level of support is a microkernel, and the support can be broadened by demands of the application. An example of this type of kernel is the Chorus microkernel [9], which can be scaled from a simple embedded microkernel to a full POSIX/UNIX support.

7.2 Real-time Extensions To Commercial Timesharing Operating Systems

The second approach to the real-time operating system is the extension of commercial products, like extending UNIX to RT-UNIX, or POSIX to RT-POSIX. The real-time versions for commercial operating systems are generally slower and less predictable than proprietary kernels, but they have a greater functionality and a better software development environments.

However, there are various problems when attempting to convert a non-real-time operating system to a real-time version [3]. For example, in UNIX interface, problems exist in process scheduling. These problems are due to nature of the nice and setpriority primitives as well as round robin scheduling policy. Furthermore, timer facilities in UNIX are too coarse, memory management is too complex, and interprocess communication facilities do not support fast and predictable communication. Also implementation issues used in the UNIX operating system restrict its

use for real-time purposes. These issues include intolerable overhead, nonpreemptability of the kernel, and internal FIFO queues. As a result, extending commercial timesharing operating system for real-time purposes is not a feasible approach, especially when hard deadlines are needed.

7.3 Research Kernels

The third category of real-time operating system is research kernels. These kernels are used in research purposes to develop and demonstrate new features to the real-time operating systems. Trends in the current research in real-time operating systems include developing real-time process models, developing real-time synchronization primitives, searching solutions for timing analysis, developing support for fault tolerance, investigating object-oriented approaches, providing support for multiprocessor as well as distribution, and attempting to formally define a microkernel. There are numerous research project addressing these issues. A survey of these projects is found in [13].

8. Designing a reflective architecture for a real-time operating system

There are several issues that has to be taken into account when designing and building a real-time operating system; meeting functional, fault tolerance and timing requirements are complex tasks. One method in building a complex and flexible real-time system is to use the notion of reflective architecture [5]. A system based on reflective architecture is one that reasons about and reflects upon its own current state and that of the environment to determine the right course of action. Several current real-time systems contain the same basic paradigms found in timesharing operating systems. They are usually only stripped down and optimized versions of time-sharing operating systems. Although they stress fast mechanisms such as fast context switching and ability to respond to external interrupts quickly, they retain some main abstractions of timesharing systems, which should be taken into account when designing a complex and flexible real-time operating system, including:

- Viewing the execution of a task as a random process where task could be blocked at arbitrary points during its execution for an indefinite time. In critical real-time environments each task is well defined and can be analyzed a priori
- Assuming that little is known about the tasks a priori, so that little semantic information about them is utilized at run time. In real-time systems, the software should be able to make use of important semantics information about the application tasks.
- Attempting to maximize throughput or minimize average response time.

These metrics are not primary metrics for real-time systems. System could have a good average response time and still miss every deadline, resulting a useless system.

In traditional systems computation solves some application problem such as sorting a file or filtering important signals from radar returns. In reflective system we can consider a system to have a

computational part and reflective part. Computational part acts like in the traditional systems, but reflective part of the system exposes the system state and semantics of the application to decide what to do, for example in filtering process to choose the most appropriate filter.

5.1 Design issues There exists also several opposing factors, that make the design even more complex. The desire for predictability versus the need for flexibility to handle non-deterministic environments, failures and system architecture, the need for efficient performance and low cost versus understandability etc. Flexible realtime system architecture cannot be based on handcraft solutions because of their complexity and requirements. When building a real-time system based on a reflective architecture means that first we must identify reflective information regarding the system. This information includes:

- Importance of task, group of tasks and how tasks' importance relates to other and to system modes.
- Time requirements (deadlines, periods etc.)
- Time profiles, such as the worst-case execution times
- Resource needs
- Precedence constraints
- Communication requirements
- Objectives or goals of the system
- Consistency and integrity constraints
- Policies to guide system-wide scheduling
- Fault tolerance requirements
- Policies to guide tradeoff analyses
- Performance monitoring information

Implementation structures in the operating system retain this information and primitives allow it to be dynamically changed. When implementing a real-time operating system based on reflective architecture, we need to use real-time programming languages. First we need a high-level programming language that is capable of specifying reflective information such as timing requirements and the need for guarantees with respect to these requirements. An example of this is Spring-C, which is used in Spring -system implementation [3]. In general, this language is C -based programming language with support for timing specifications and guarantee semantics. Then we need a language to support system description. That should provide means to perform careful, detailed and accurate timing analysis. Third, we require notation for specifying fault-tolerance requirements on a task-by-task basis, where it is possible to perform adaptive management of redundancy.

When tasks' priorities are fixed, a fixed-priority scheduling mechanism, provided by most real-time kernels, works properly. Still many more complex systems require dynamic priorities. Dynamic priorities can be dynamically calculated for example with some weighted formula that combines different information e.g. deadline and resource needs. Systems with fixed priorities are not able to handle dynamic priorities properly.

In order to deal with predictability versus flexibility, there is a need for multilevel, multi-dimensional scheduling algorithms [5]. These algorithms explicitly categorize the performance, including when there is system degradation or unexpected events. Algorithms are multi-dimensional, because they have to consider all resources needed by a task, not just CPU time, and they must consider precedence constraints and communication requirements. They are multi-level algorithms, because tasks are categorized into four classes; critical (missing the deadline causes a total system failure), essential (these tasks have hard deadlines, but the tasks are not critical), soft real-time (task value drop after deadline but not to zero or negative number), and non-real-time (no deadlines). When task arrives, scheduling algorithm uses the reflective information about active tasks and creates a full schedule for them and predicts if some of them is going to miss their deadlines. If deadlines would be missed, error handling can occur before that happens.

Many real-time systems require strict fault tolerance in order to operate in complex, highly variable environment. A three-level framework for defining fault tolerance requirements is presented. At the highest level of the framework is the overall design process. That includes specification of the physical inputs and outputs from and to the external world, first specification of the important functional blocks in the system, the flow of data and interactions among them, a definition of timing requirements (periods, deadlines and response times), identification of each functions criticality, and specification of mode changes. At the second level of the framework redundancy of the system is managed. On third level of the framework is the actual coding of the application modules themselves. These three levels must be consistent with each other.

9. 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

Real-time systems

www.bestneo.com

management support, transactionality, failure recovery mechanisms,
data temporality, and real-time access to data.