REAL-TIME SYSTEMS: AN INTRODUCTION AND THE STATE-OF-THE-ART

INTRODUCTION

Our goal in this article is to give an overview of the broad area of real-time systems. This task daunting because real-time systems are everywhere, and yet no generally accepted definition differentiates real-time systems from non–real-time systems. We will make an attempt at providing a general overview of the different classes of real-time systems, scheduling of tasks (or threads) in such systems, design tools and environments for real-time systems, real-time operating systems, and embedded systems. We will conclude our discussions with research challenges that still remain.

Definitions

Real-time systems cover a broad spectrum of automated platforms in which the correctness of the system not only requires functionally (or logical) correct operation but also produces results within prespecified "real-time constraints." By contrast, a *non–real-time system* is one for which there is no deadline, even if fast response or high performance is desired or even preferred. Real-time systems are usually "situated" in an environment and involve in sensing apparatus to detect, control, and adapt to the environmental conditions. Real-time systems can be networked and distributed (e.g., sensor networks) or embedded (e.g., automotive control systems, cell phones).

Hard versus Soft Real-Time Systems

Real-time systems are classified as hard or soft real-time systems. *Hard real-time systems* have very strict time

Wiley Encyclopedia of Computer Science and Engineering, edited by Benjamin Wah.

constraints, in which missing the specified deadline is unacceptable. The system must be designed to guarantee all time constraints. Every resource management system such as the scheduler, input–output (I/O) manager, and communications, must work in the correct order to meet the specified time constraints.

Military applications and space missions are typical instances of hard real-time systems. Some applications with real-time requirements include telecom switching, car navigation, the medical instruments with the critical time constraints, rocket and satellite control, aircraft control and navigation, industrial automation and control, and robotics.

Soft real-time systems also have time constraints; however, missing some deadline may not lead to catastrophic failure of the system. Thus, soft real-time systems are similar to hard real-time systems in their infrastructure requirements, but it is not necessary that every time constraint be met. In other words, some time constraints are not strict, but they are nonetheless important. A soft real-time system is not equivalent to non-real-time system, because the goal of the system is still to meet as many deadlines as possible.

Some applications with soft real-time requirements include web services such as real-time query, call admittance in voice over internet protocol and cell phone, digital TV transmissions, cable and digital TV set-top-boxes, video conferencing, TV broadcasting, games, and gaming equipment. Multimedia systems in general are examples of soft real-time systems (e.g., dropping frames while displaying video).

Even in some typical hard real-time applications, some functions have soft real-time constraints. For instance, in Apollo 11, the lunar module guidance computer could not keep up with the data stream from the landing radar. However, it was discovered that the missed deadlines were nonfatal, and the scheduler automatically adjusted, to meet soft real-time behavior for the landing tasks.

Periodic and Aperiodic Tasks

Real-time applications are also classified depending on the tasks that comprise the application. In some systems, tasks are executed repetitively within a specified period. A task t_i is characterized as (p_i, e_i), where p_i its periodicity and e_i is its (worst-case) execution time. Monitoring patient's vitals is an example of such a system. Hard real-time systems are usually designed using periodic tasks, and static scheduling can be used for periodic tasks.

Aperiodic tasks are tasks that are executed on demand (or with unknown period). A task is executed in response to an event, and a task t_i is characterized as (a_i, r_i, e_i, d_i) where a_i its the arrival time. r_i is the time when the task is ready for execution, e_i is its (worst-case) execution time, and d_i is the deadline by which the task must complete. It should be noted that the arrival time may not be specified in some systems, and the ready time is defined by the arrival of an event. Real-time systems that must react to external stimuli will consist of aperiodic tasks, which define the response to the events. Systems that include aperiodic tasks fall into the class of soft real-time applications, because scheduling may not guarantee completion of tasks within specified deadlines.

Real-time applications may also include sporadic tasks, which are defined random arrival times. Sporadic task can be characterized by (a_i, r_i, e_i, d_i); similar to aperiodic tasks.

Embedded Real-Time Systems

Hard real-time systems typically interface with the physical hardware at a low level in an embedded system. The embedded system is usually a special-purpose system designed to perform a few or even only one dedicated function usually with real-time computing constraints. Antilock brakes on a car is a simple example of an embedded real-time system in which the real-time constraint is the short time in which the brakes must be released to prevent the wheel from locking. Other examples include medical systems such as heart pacemakers, industrial process controllers, communication systems, aircraft control systems, and weapon systems.

In general, embedded real-time systems are designed as reactive systems: The system observes changes in the environment, computes appropriate actions, and conveys the actions to various components so that the system as a whole operates correctly while the designated time constraint is met. For example, cruise-control systems in automobiles must continuously monitor and react to current speed of the vehicle and adjust the engine's acceleration or deceleration appropriately within a prespecified time; long delays will cause the system to fail in maintaining the cruising speed. In addition to meeting correctness and timing constraints, the embedded real-time systems must also meet constraints imposed by the embedded nature of such systems. These constrain include (1):

1. Nonrecurring engineering costs: The one-time cost of designing system must be as minimal as possible.
2. Unit Cost: The cost of each unit must be minimal.
3. Size: The physical dimension of the system is limited by the environment in which it will be embedded.
4. Power: Because some embedded systems run on batteries, various power management schemes must be employed to minimize the power consumption.
5. Performance: If the system must meet real-time constraints, then performance is a critical metric.

In many cases, embedded real-time systems use application-specific hardware [such as Digital Signal Processing (DSP) processors and application specific integrated circuits (ASIC)]. However, more recently, general-purpose processors along with reconfigurable fabrics [i.e., field programmable gate arrays (FPGAs)] are becoming commonly used to designing real-time embedded systems. The reconfigurable fabric can be used to speed up common functionalities without having to custom design circuits. The reconfigurability allows flexibility to support multiple functionalities and added features. Recent technological innovations have led to very powerful general-purpose processors that can be used to meet the performance requirements of modern embedded real-time systems. Such systems also permit for power saving options such as dynamic frequency

and voltage control, shutting unused subsystems, and reconfigurability of memory [particularly cache (2–5)] subsystems.

Several design tools are currently available either to customize the hardware and software subsystems or to use general-purpose processors augmented with FPGA components. The nonrecurring costs are higher with the first option (ASIC); however, such systems can be designed to meet stringent size, power, and performance requirements. The latter option (FPGA) provides greater flexibility.

These tools, however, are designed for use by experts with a clear understanding of both hardware and software. Tools are needed that permit higher-level abstractions to designers and that automatically optimize lower-level systems to meet the specified constraints.

Cyber-Physical Systems

Cyber-physical systems (6) are integrations of physical processes with computation. Networks and embedded computers monitor and control the physical processes using feedback loops where computations affect physical processes and vice versa. Applications include medical devices, traffic controllers, advanced automotive systems, avionics, critical infrastructure control (e.g., electric power, water resources, and communication systems), defense systems, and smart structures. Cyber-physical systems are concurrent by nature, because physical processes are intrinsically concurrent, and their coupling with computing requires concurrent composition of the computing processes with the physical system.

SCHEDULING IN REAL-TIME SYSTEMS

Most real-time systems are designed as concurrent processing systems, rather than as monolithic control systems. The concurrency allows for the system to react to events more easily. The scheduling of concurrent activities (tasks or threads) is critical to achieving real-time constraints. Different scheduling approaches are available for different types of real-time systems: hard versus soft real-time; periodic, aperiodic or sporadic tasks. Most scheduling algorithms aim to meet deadlines associated with tasks while optimizing the use of resources.

Static versus Dynamic Priority Scheduling

Classic scheduling theory deals with static scheduling. Static scheduling refers to the fact that the scheduling algorithm has complete information regarding the task set, which includes knowledge of deadlines, execution times, precedence constraints, and release times.

In rate-monotonic (RM) scheduling, the shorter the period of a task, the higher is its priority. In deadline-monotonic (DM) scheduling, the shorter the relative deadline (i.e., the difference between the deadline and the current time, also known as the laxity) of a task, the higher is its priority. This approach investigates schedulability tests for sets of periodic tasks whose deadlines are permitted to be less than their period. Such a relaxation enables sporadic tasks to be incorporated directly with periodic tasks (7, 8). For arbitrary relative deadlines, DM outperforms RM in terms of use.

Real-Time Algorithm Metrics

The most important metric of a real-time system is the success ratio of system deadlines. The success ratio is defined as the percentage of jobs completed before their deadlines. However, other metrics, such as the minimized total (or weighted sum) of the execution times of real-time jobs, the minimized average response time, the minimized maximum lateness or tardiness of real-time jobs, and the minimized number of processors required for real-time jobs, may be important for real-time systems, especially for soft real-time systems.

In soft real-time systems, missing a few deadlines is not critical; however, the overall performance and use of resources are important. Using the aforementioned metrics (in addition to success ratios) is often overlooked in many real-time systems. Minimizing total or average execution time has secondary importance in helping to minimize resource requirements for a system. It should be noted that minimizing execution time does not directly address the fact that individual tasks have deadlines.

For instance, minimizing the maximum lateness metric can be useful during the design process where resources can be continually added until the maximum lateness ≤ 0 (i.e., no deadline is missed) is met. However, generally, the metric is not always useful because minimizing the maximum lateness does not necessarily prevent tasks from missing their deadlines.

In some applications, the "quality of service" (QoS) is used as a metric in achieving specified requirements. QoS requirements allow for trading off different performance parameters. For example, in some applications, users may be "charged" different rates depending on the level of the service required. The objective in such systems is the optimization of total revenues by allocating resources to tasks of users who pay higher rates.

Another concept that often appears in the real-time literature is the optimality of an algorithm. Scheduling algorithm is optimal if no other scheduling algorithm can find a better solution for the same scheduling problem.

Scheduling Algorithms

Real-time systems are also distinguished based on their implementation: In preemptive systems, tasks may be preempted by higher priority tasks, whereas nonpreemptive systems do not permit preemption. It is easier to design preemptive scheduling algorithms for real-time systems. However, nonpreemptive scheduling is more efficient, particularly for soft real-time applications and applications designed for multithreaded systems, than the preemptive approach caused by the reduced overhead needed for switching among tasks or threads (9,10).

In many practical real-time scheduling scenarios such as I/O scheduling, the properties of device hardware and software make preemption either impossible or prohibitively expensive. Nonpreemptive scheduling algorithms are easier to implement than preemptive algorithms, and they can exhibit dramatically lower overhead at run time. The

overhead of preemptive algorithms is more difficult to characterize and predict than that of nonpreemptive algorithms. Nonpreemptive scheduling on a uniprocessor naturally guarantees exclusive access to shared resources and data, which eliminates both the need for synchronization and its associated overhead. The problem of scheduling all tasks without preemption forms the theoretical basis for more general tasking models that include sharing of resources.

Scheduling decisions can be clock driven or priority driven. In the first case, scheduling decisions are made at specific time instants, and these instants are chosen a priori. Hard real-time systems use this approach. In a priority-driven (or event-driven), approach scheduling decisions are made when tasks complete or resources become available (which permits other tasks to acquire the resources and start execution).

Earliest Deadline First (EDF) Scheduling. The EDF algorithm is the most widely used scheduling algorithm for real-time systems on uniprocessors and multiprocessors (11,12). Recall that real-time applications can be characterized as hard real-time or soft real-time systems. Hard real-time applications require that all time constraints be met, whereas soft real-time systems permit some tolerance in meeting time constraints.

For a set of preemptive tasks (be periodic, aperiodic, or sporadic), EDF will find a schedule if a schedule is possible (13). The application of EDF for nonpreemptive tasks is not as widely studied. EDF is optimal for sporadic nonpreemptive tasks, but EDF may not find an optimal schedule for periodic and aperiodic nonpreemptive tasks.

EDF scheduling is one of the first dynamic priority-driven scheduling algorithms proposed. As the name implies, tasks are selected for execution in the order of their deadlines. It provides the basis for many real-time algorithms. EDF suffers significantly when the system is overloaded. Compared with static priority-driven scheduling such as RM with approximate 69% use, EDF can approach 100% use for periodic jobs.

It should be noted that dynamic scheduling does not mean online scheduling. An online-scheduling algorithm has only complete knowledge of the currently active set of tasks, and no knowledge of any new arriving tasks. Likewise, offline scheduling is not the same as static scheduling. Offline includes preanalysis of scheduling regardless of whether the runtime algorithm is static or dynamic. Usually, offline scheduling has higher performance than online scheduling but may lead to poorer use of resources.

First Come First Served (FCFS) Scheduling. FCFS scheduling uses a simple "first in, first out" (FIFO) queue. It is simple to implement, but it has several deficiencies. Its average wait time is typically long relative to EDF. It is a nonpreemptive scheduling technique, and it is subject to the negative effect in I/O-bound applications. In such cases, large amounts of idle times can occur as the I/O-bound processes sit idle waiting for the Central Processing Unit (CPU)-bound process to complete.

Round Robin (RR) Scheduling. RR scheduling is similar to FCFS scheduling with added preemptive capability. As the name suggests, on each time quantum a new process receives access to the system resources. This way, each process gets a share of the system resources without having to wait for all processes ahead of it to run to completion. The average wait time is typically long relative to EDF, and its performance is proportional to the size of the time quantum. RR scheduling is the degenerative case of priority scheduling when all priorities are equal.

Shortest Job First (SJF) Scheduling. SJF scheduling is probably optimal but requires clairvoyance, profiling, or expected execution time to implement fully. SJF can be implemented either preemptively or nonpreemptively. SJF has low average waiting time. In fact, SJF is optimal with respect to average waiting time.

Although FIFO/FCFS, RR, and SJF are very basic real-time scheduling schemes, they are widely implemented in real-time systems.

Group-EDF (gEDF) Scheduling. gEDF is a variation of EDF that can improve the success ratio (that is, the number of tasks that have been successfully scheduled to meet their deadlines), particularly in overloaded conditions. gEDF can also decrease the average response time for tasks. In gEDF, tasks with "similar" deadlines are grouped together (i.e., deadlines that are very close to one another), and the SJF algorithm is used for scheduling tasks within a group. It should be noted that this approach is different from adaptive schemes that switch between different scheduling strategies based on system load; gEDF is used in overloaded as well as underloaded conditions. The computational complexity of gEDF is approximately the same as that of EDF.

gEDF is particularly useful for soft real-time systems as well as applications known as "anytime algorithms" and "approximate algorithms," in which applications generate more accurate results or rewards with increased execution times (14,15). Examples of such applications include search algorithms, neural-net based learning in artificial intelligence, fast fourier transform, and block-recursive filters used for audio and image processing.

Table 1 shows the taxonomy of real-time scheduling in real time systems.

Table 1. The Taxonomy of Real-Time Scheduling in Real-Time Systems

Soft	Dynamic	Nonpreemptive
		Preemptive
	Static	Nonpreemptive
		Preemptive
Hard	Dynamic	Nonpreemptive
		Preemptive
	Static	Nonpreemptive
		Preemptive

DESIGN AND ANALYSIS TOOLS AND ENVIRONMENTS

Real-time systems interact with real-world entities, and the interactions can be very complex. The system must respond to real-world events within a specified time. Because many real-time systems are embedded, the designers must deal with both hardware and software subsystems. When multiple processors are involved, it is also necessary to decide on the interconnections among the processors and the speed of communication among the processing nodes. Communication and task scheduling is often performed by the operating system (OS) running on the processing nodes. Thus, it is necessary to understand the OS overheads while estimating execution times and message communication delays.

Real-time design tools should permit the representation of concurrency, communication, synchronization, and timing constraints. Additionally, the tools should facilitate analysis of the temporal characteristics for feasibility, predictability, and correctness. A typical design workflow of real-time system is shown in Fig. 1 taken from Refs. 16 and 17. The shaded boxes represent activities providing feedback that determines the feasibility of the design.

Since Unified Modeling Language (UML) (18) has received widespread acceptance among software designers, several UML based design methodologies for real-time systems are becoming available. Several real-time design patterns are available, which can aid in the real-time software design. Examples of patterns include communication patterns, resource management patterns, patterns for hierarchical state machines, hardware interface patterns, and architectural patterns.

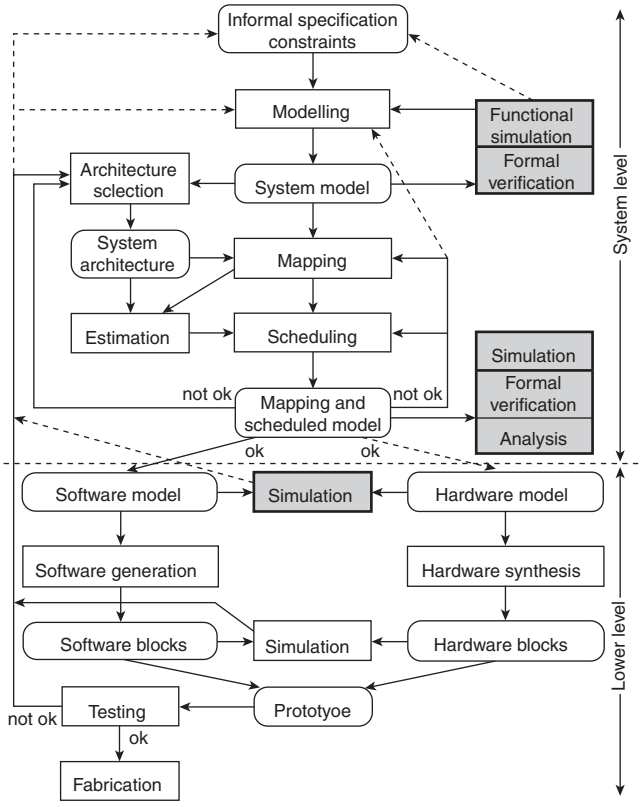

Figure 1. Design workflow of real-time system.

In this section, we will discuss the Harmony design process (19). The process starts with a system functional analysis with use-case models. The Harmony process permits incremental development of the use case model. System architectural design is the next step in the Harmony process. In this step, the overall system architecture is defined, which includes functional blocks along with their connections and interfaces. It should be noted that system blocks include both hardware and software components, and it is necessary to model them for analysis so that the system can be verified. Harmony permits incremental modeling and analysis as well as formal specifications and analysis using state machines. Following the model-based analysis, Harmony allows the development of a design for the system. This phase allows optimization of the system for implementation. The optimization at architectural level considers subsystems, concurrency, distribution of tasks, deployment, and QoS (safety, reliability, and possibly security). The mechanistic design phase of Harmony is concerned with the optimization of collaborations. Finally, the detailed design phase elaborates the implementation of the objects and classes, which include data structures and algorithms. The last two steps of the Harmony process involve the actual implementation (code generation) and testing.

Harmony allows for a systematic approach to the design of software for real-time systems. It should be remembered, however, that real-time systems include not only software but also hardware and mechanical systems. The timing analysis for tasks must take into account for OS overheads, unless the system is small and OS services are included in the design. Most CPUs used in embedded systems are becoming complex, and modeling execution times on such systems is very difficult, particularly in the presence of components that exhibit nondeterministic behaviors (e.g., Cache memories).

Because real-time systems include multiple tasks or threads, testing such systems is difficult as the interactions among the threads may not be reproducible. Improper use of synchronization (such as mutual exclusion) can either lead to performance bottlenecks or race conditions. Additional capabilities are needed with Harmony to aid the design of complex real-time systems fully.

Predictability and Schedulability Analysis

Schedulability analysis is used to verify whether every task in the real-time system can meet its deadline when scheduled on the deployed system. The precise (and predictable) execution time analyses are needed for hard real-time systems. RM (20,21) and DM (22,23) are fixed-priority scheduling algorithms, which are widely used for hard real-time systems. These algorithms require exact or worst-case execution times of individual tasks. The execution times can be estimated from actual execution of the task or the analysis of the code. Several methods are available for estimating execution time of tasks and designing languages that permit predictable execution times for tasks. For example, task execution

times can be estimated from instruction counts and maximum number of loop iterations. To obtain worst-case execution estimates, the task is analyzed for longest control paths and without cache memories.

Petri nets have been used to model and analyze real-time systems. A Petri net is a bipartite graph with two types of nodes called Places and Transitions, which represent events and activities. The occurrence of an event is represented by the presence of a token in the corresponding place. An activity (transition) is enabled with all input places that contain tokens (representing the occurrence of input events), and when the transaction completes execution, it stores tokens in all its output places. Time can be associated either with a transition (Timed Transition Petri nets) or with places (Timed Place Petri Nets). Execution times may also be represented as probability distributions that lead to stochastic Petri nets. Resources can be modeled as places.

For soft real-time systems, exact execution times are not required. Execution times can be modeled as probability distributions. Schedulability analysis then will provide a probabilistic estimate of meeting the time specifications.

Performability and Stochastic Assurance

The term "performability" was first introduced by Meyer et al. (24) to combine performance and reliability analyses in fault-tolerance systems. Informally, performability can be defined as the probability that a system performs at different levels of "accomplishment." We can extend the definition of performability to real-time systems and define the term as the probability that the set of tasks that comprise the real-time system complete their execution successfully by the deadline defined with the system. The failure of a task to complete successfully within the specified deadline is included in the computation of performability. A failure to meet deadlines may be because of the following reasons.

1. For a particular instance, the input data required longer execution time, which is particularly true with systems that include cache memories, branch prediction and other techniques that are not easy to include in task execution time estimations. This is also the case when a choice between migrating the task or data versus executing the task at the site where it is initiated lead to different execution times for the task.
2. Because of failures, the task has to be re-executed, migrated, and restarted.
3. The task is delayed (hence missed the deadline) because the failure of a preceding task to complete successfully.

If P_{D_i} is the probability that task J_i will complete successfully during its deadline D_i, then it is possible to obtain the probability of the system (i.e., performability) comprising several tasks in meeting system deadlines. This probability can be improved by providing the system with some "slack" time so that tasks failing to meet deadlines are provided with additional time to complete, and yet meet the overall deadline.

REAL-TIME OPERATING SYSTEMS (RTOS)

Relatively very few and simple real-time systems can be developed effectively as a single control program. Most real-time systems use a RTOS or a kernel to support at a minimum creation and management of tasks (and threads) and scheduling of the task on available processing resources. A real-time kernel is memory resident part of an operating system that provides the necessary services to real-time applications. Many RTOS and kernels are available, which include ThreadX from Xpresslogic, VxWorks by Wind River, QNX Neutrino by QNX Software Systems, Real-time Linux (by several vendors). A key characteristic of a RTOS is that all services (or system calls) must have deterministic behavior. In addition, the response time to interrupts must have a guaranteed worst-case latency, and context switching times must be very short.

RTOS (25) is not simply a real-time system. It is the core part of any real-time system. A real-time system includes all the system elements such as hardware, middleware, applications, communications, and I/O devices. All the elements are needed to meet the system requirements. However, RTOS provides sufficient functionality to enable a real-time application to meet its requirements. It is also important to distinguish between a fast operating system and a RTOS. Speed, although useful for meeting the overall requirements, by itself is not sufficient to determine whether a system meets the requirements for an RTOS.

Real-time OS kernels are either designed as monolithic kernels or micro-kernels. In the first case, the OS kernel provides all required services (including I/O, scheduling, memory management). Applications make calls to these services. Such designs are easy to design and implement, but they are generally not suitable for larger systems. In the case of a micro-kernel, the system includes very basic services. Other system services run as separate user processes. Applications invoke these services (via message passing) and the microkernel arbitrates the request for the services. Microkernels are easier to maintain in large system and incur overheads because of message passing and switching between applications and system services.

Real-time OS systems often support very limited virtual memory functionality. For example, RT-Linux keeps all real-time applications in a dedicated address space; this memory is never paged out of main memory. RTOS normally support several scheduling methods and applications have the flexibility of specifying which scheduling to use. Most RTOS systems support POSIX-defined communication and synchronization mechanisms (e.g., semaphores, mutexes, condition variables, spin locks, signals, pipes, and message queues).

Table 2 compares some commonly available RTOSs.

POSIX 1003.1 for RTOS

The IEEE Computer Society's Portable Application Standards Committee defined a standard for Portable Operating System Interface (POSIX) (26,27). This IEEE Standard 1003.1 includes IEEE Standard 1003.1a, IEEE Standard 1003.1b, and 1003.1c; IEEE Standard 1003.1d/j/q; and IEEE Standard 1003.13. IEEE Standard 1003.1a is the base for all the POSIX standards. IEEE Standard

Table 2. Comparison of Real-Time Operating Systems

	Kernel architecture	IPC	Memory management	File systems	Development environment
RTLinux	Monolithic	POSIX	No virtual memory	Ext3	No tool chain
VxWorks	Microkernel	POSIX	Virtual memory	DOS, NTFS, RT11	Tornado tool chain
QNX	Microkernel	POSIX	Virtual memory	DOS	Eclipse tools
WinCE	Monolithic	Non-POSIX	Restricted virtual memory support	FAT, RAMFS	Embedded Visual C++ tools
LynxOS	Microkernel	POSIX	Virtual memory	Flash, RAMFS	Visual Lynx
TinyOS	Monolithic	Non POSIX	No virtual memory	Flash	NesC

1003.1b (formerly POSIX 1003.4) defines the needed real-time extensions. IEEE Standard 1003.1c defines the functionality of threads. These various standards have been combined by the Austin Group in producing IEEE standard 1003.1-2001. The latest version is now known as the IEEE 1003.1 2004 Edition.

POSIX 1003.1b provides the standard criteria for RTOS services and is designed to allow programmers to write applications that can easily be ported to any OS that is POSIX compliant. The basic RTOS services covered by POSIX 1003.1b include asynchronous I/O, synchronous I/O, memory locking, semaphores, shared memory, timers, inter-process communication, real-time files, real-time threads, and scheduling.

Real-time scheduling is the most important feature of a RTOS. POSIX 1003.1b specifies the following scheduling policies.

SCHED FIFO - Priority based preemptive scheduling, FIFO is used among tasks with the same priority.

SCHED RR - Processes with same priority use Round Robin policy. A process executes for a quantum of time; and then it is moved to the end of the queue corresponding to its priority level. Higher priority tasks can preempt tasks. The size of the quantum can be fixed, configurable, or specific for each priority level.

SCHED OTHER - Availability required but not defined by the standard. Usually SCHED OTHER is implemented as a classic time-sharing policy.

RTOS Examples

Microsoft Windows CE—Non-Linux Based Commercial RTOS. Microsoft Windows CE is designed as a general-purpose and portable real-time operating system for small memory, 32-bit mobile devices. Windows CE slices CPU time among threads and provides 256 priority levels. To optimize performance, all threads are enabled to run in kernel mode. All nonpreemptive portions of the kernel are broken into small sections, which reduces the duration of non-preemptive code. Windows CE incurs long latencies for tasks.

VxWorks—Commercial RTOS. VxWorks, by Wind River Systems (Almeda, CA), is a real-time operating system. It runs currently on its own kernel. However, its development is done on a host machine such as Linux or Windows. Its cross-compiled target software can be run on various target CPU architectures. VxWorks runs in supervisor mode, and it does not use traps for system calls. VxWorks supports priority interrupt-driven preemption and optional round-robin time slicing. The micro kernel supports 256 priority levels. VxWorks supports some IEEE POSIX 1003.1 functions.

LynxOS—POSIX Compatible Commercial RTOS. LynxOS is a POSIX compatible, multithreaded OS designed for complex real-time applications that require fast, deterministic response. It is scalable from small, embedded products to large switching systems. The microkernel can schedule, dispatch interrupts, and synchronize tasks. It uses scheduling policies such as prioritized FIFO, Dynamic Deadline Monotonic (DDM, the shorter the dynamic deadline, the higher is its priority) scheduling, time-slicing, and so on. It has 512 priority levels and supports remote console and remote monitoring. For instance, LynxOS can be used as a hard real-time system for controlling gas levels in chemical plants remotely.

RTLinux—Open Source Linux-Based RTOS. RTLinux is a hard real-time operating system that runs Linux as its lowest priority thread. The Linux thread is completely preemptible so that real-time threads and interrupt handlers are never delayed by non-real-time operations. Real-time applications can make use of all the powerful, non-real-time services of Linux. RTLinux scheduling policies supports EDF. RTLinux, which was originally developed at the New Mexico Institute of Technology, is an open-source product. RTLinux-specific components are released under the GNU General Public License (GPL), and Linux components are released under the standard Linux license. The source code is freely distributed. Non-GPL versions of the RTLinux components are available from FSMLabs (28).

RED-Linux—Open Source Linux-Based RTOS. RED-Linux (29) is an open-source real-time and embedded version of Linux version 2.2.14. In addition to the original Linux capability, it improves the real-time behaviors of the Linux kernel in many ways. RED-Linux supports a short kernel blocking time, a quick task response time, and modularized runtime General Scheduler Interface so that different scheduling methods can be selected depending on the application.

Kurt-Linux—Open Source Linux-Based RTOS. KU Real-Time Linux (KURT) (30) is a Linux system with real-time modifications that allow scheduling of real-time events at tens of microseconds resolution. Rather than relying on priority-based scheduling or strictly periodic schedules, KURT relies on application-specified schedules. KURT can function in two modes: focused mode, in which only real-time processes are allowed to run; and mixed mode, in which the execution of real-time processes still takes precedence, but non-real-time processes are allowed to run when real-time tasks are not running. KURT was developed by the Information and Telecommunication Technology

Center at the University of Kansas. KURT may be used and distributed according to the terms of the GNU Public License.

QNX—Commercial POSIX-compliant Unix-like RTOS. QNX is a commercial POSIX-compliant Unix-like real-time operating system, which is aimed primarily at embedded systems. The QNX kernel contains only interprocess communication, CPU scheduling, interrupt redirection, and timers. Memory management by operating in conjunction with the microkernel runs as user process. Because of the microkernel architecture, QNX is also a distributed OS known also as transparent distributed processing. All I/O operations, network operations, and file system operations work through message passing, which includes data transfer. The source for QNX kernel has been released for noncommercial use.

CONCLUSIONS

In this article, we attempted to provide a broad overview of real-time systems. It is hoped that the reader gains a general appreciation of the range of applications, different types of requirements that real-time systems must meet, and the software and hardware design tools that are available. However, many challenges still remain before real-time systems can be implemented without very specialized expertise or hand coding of applications to meet timing constraints.

Our society is becoming more automated, and we are discovering that many real-world systems have real-time constraints. These constraints include transportation systems, financial systems, medical and healthcare systems, intelligent residential and commercial workplaces, and services provided by public and private organizations. Many of these systems will be interconnected, which complicates the design and implementation of such systems. It will become necessary to include dynamic adaptability so that systems can adapt to unexpected events. Dynamic adaptation implies continuous monitoring to detect for unexpected situations and to provide proper response from the system (either automatically or with the aid of a human).

Lee (6) advocates a top-to-bottom rethinking of real-time and cyber physical systems. However, this method may be incompatible with legacy systems that may have to be integrated with newer environments.

Simulation languages such as Simulink (31) have permitted the specification of time. Newer programming languages [e.g. Giotto (32)] permit the specification of timing semantics and reasoning about real-time systems. Such time specifications should be integral to any programming language.

Another recommendation of Lee (6) is that the traditional boundary between operating systems and programming languages should be eliminated. Applications should specify the needed services, and the system should be assembled with only the required services.

Design environments should permit the adaptation of applications to different hardware and software platforms, which will require the estimation of execution times (and other parameters as energy consumption, reliability, communication delays) for each environment. Hardware/software codesign environments must be enhanced with such capabilities. The tools must also be easy to use.

BIBLIOGRAPHY

1. F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software, Introduction*. New York: Wiley, 2002.
2. A. Gordon-Ross, F. Vahid, and N. Dutt, Automatic tuning of two-level caches to embedded applications *Proc. Design Automation and Test in Europe Conference (DATE)*, February 2004.
3. C. Zhang, F. Vahid, and W. Najjar, A highly configurable cache architecture for embedded systems, *Proc. 30th Annual International Symposium on Computer Architecture*, June 2003, pp. 136–146.
4. A. Naz, K. Kavi, W. Li, and P. Sweany, Tiny split data caches make big performance impact for embedded applications, *J. Embed. Comput.* **2**(2): 207–219, 2006.
5. A. Naz, K. Kavi, J. Oh, and P. Foglia, Reconfigurable split data caches: A novel scheme for embedded systems, *Proc. 22nd Annual ACM Symposium on Applied Computing*, Seoul, Korea, March 11–15, 2007, pp. 707–7112.
6. E. Lee, Cyber-physical systems—are computing foundations adequate? *Proc. NSF Workshop on Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, Austin, TX, October. 2006.
7. N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings Hard real-time scheduling: the deadline-monotonic approach, *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, 1991.
8. L. Sha, R. Rajkumar, and S. S. Sathaye, Generalized rate-monotonic scheduling theory: A framework for developing real-time systems, *Proc. IEEE*, January. 1994.
9. R. Jain, C. J. Hughes, and S. V. Adve, Soft real-time scheduling on simultaneous multithreaded processors, *Proc. 23rd IEEE International Real-Time Systems Symposium*, December 2002.
10. K. M. Kavi, R. Giorgi, and J. Arul, Scheduled dataflow: Execution paradigm, architecture, and performance evaluation, *IEEE Trans. Comp.*, **50**, 2001.
11. F. Balarin, L. Lavagno, P. Murthy, and A. S. Vincentelli, Scheduling for embedded real-time systems, *IEEE Design Test Comp.*, January–March 1998.
12. J. H. Anderson, V. Bud, and U. C. Devi, An EDF-based scheduling algorithm for multiprocessor soft real-time systems, *Proc. 17th Euromicro Conference on Real-Time Systems*, 2005.
13. C. L. Liu and J. W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, *J. ACM*, **20**(1): 46–61, 1973.
14. J. K. Dey, J. Kurose, and D. Towsley, Online Processor Scheduling for a Class of IRIS (Increasing Reward with Increasing Service) Real-Time Tasks, Tech. Rep. 93–09, Department of Computer Science, University of Massachusetts, Amherst, January 1993.
15. S. Zilberstein, Using anytime algorithms in intelligent systems, *AI Mag.*, pp.71–83, Fall 1996.
16. P. Eles, System Design and Methodology, Tech. Rep., Linkoping University, Sweden. Available: http://www.ida.liu.se/~TDTS30/, 2002.
17. S. Manolache, Analysis and optimization of real-time systems with stochastic behavior, PhD Dissertation, Sweden: Linkoping University, December 2005. Available: (http://www.artes.uu.se/publications/sorma_lic.pdf).

18. UML is defined by the OMG. Available: http://www.omg.org/.
19. B. P. Douglass, *Real-Time UML Workshop for Embedded Systems.*, Oxford:Elsevier, 2007.
20. C. L. Liu and J. W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, *J. ACM* **20**(1): 46–61, 1973.
21. J. P. Lehoczky, L. Sha, and Y. Ding, The rate monotonic scheduling algorithms: exact characterization and average behavior, *Proc. 10th IEEE Symposium on Real-Time Systems*, December, 1989, pp. 166–171.
22. N. C. Audsley, A. Burns, M. Richardson, and A. Wellings, Hard real-time scheduling: The deadline-monotonic approach, *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, May 1991, pp. 133–137.
23. N. C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, Applying new scheduling theory to static priority preemptive scheduling, *Softw. Eng. J.*, **8**(5): 284–292, 1993.
24. J. F. Meyer, D. G. Furchgott, and L. T. Wu. Performability evaluation of the SIFT Computer, *IEEE Trans. Comput.*, **C-29**, 501–509, 1980.
25. S. Baskiyar, N. Meghanathan, A survey of contemporary real-time operating systems, *Informatica*, **29**: 233–240, 2005.
26. IEEE Information Technology—Portable Operating System Interface (POSIX)—Part 1: Base Definitions; Part 2: System Interfaces; Part 3: Shell and Utilities; Part 4: Rationale. Available: http://standards.ieee.org/catalog/olis/posix.html.
27. IEEE Information Technology—Portable Operating System Interface (POSIX): IEEE/ANSI Std 1003.1, 1996.
28. Available: http://www.fsmlabs.com.
29. K. Lin, Y. C. Wang, The design and implementation of real-time schedulers in red-linux, *Proc. IEEE*, **91**(7), July 2003.
30. W. Dinkel, D. Niehaus, M. Frisbie, and J. Woltersdorf, *KURT-linux user manual*, Information and Telecommunication Technology Center, University of Kansas, 2002.
31. Available: http://www.mathworks.com.
32. T. A. Henzinger, B. Horowitz, and C. M. Kirsch, Giotto: A time-triggered language for embedded programming, *Proc. EMSOFT 2001*, Vol. LNCS 2211, Tahoe City, CA, 2001.

KRISHNA KAVI
ROBERT AKL
University of North Texas
Denton, Texas
ALI HURSON
Missouri University of Science and Technology
Rolla, Missouri