11 EXAMPLES OF REAL-TIME SYSTEMS

11.1 INTRODUCTION

Current operating systems having real-time characteristics can be divided into three main categories:

1. Priority-based kernel for embedded applications,
2. Real-time extensions of timesharing operating systems, and
3. Research operating systems.

The first category includes many commercial kernels (such as VRTX32, pSOS, OS9, VxWorks, Chorus, and so on) that, for many aspects, are optimized versions of timesharing operating systems. In general, the objective of such kernels is to achieve high performance in terms of average response time to external events. As a consequence, the main features that distinguish these kernels are a fast context switch, a small size, efficient interrupt handling, the ability to keep process code and data in main memory, the use of preemptable primitives, and the presence of fast communication mechanisms to send signals and events.

In these systems, time management is realized through a real-time clock, which is used to start computations, generate alarm signals, and check timeouts on system services. Task scheduling is typically based on fixed priorities and does not consider explicit time constraints into account, such periods or deadlines. As a result, in order to handle real-time activities, the programmer has to map a set of timing constraints into a set of fixed priorities.

Interprocess communication and synchronization usually occur by means of binary semaphores, mailboxes, events, and signals. However, mutually exclusive resources are seldom controlled by access protocols that prevent priority inversion; hence, blocking times on critical sections are practically unbounded. Only a few kernels (such as VxWorks) support a priority inheritance protocol and provide a special type of semaphores for this purpose.

The second category of operating systems includes the real-time extensions of commercial timesharing systems. For instance, RT-UNIX and RT-MACH represent the real-time extensions of UNIX and MACH, respectively.

The advantage of this approach mainly consists in the use of standard peripheral devices and interfaces that allow to speed up the development of real-time applications and simplify portability on different hardware platforms. On the other hand, the main disadvantage of such extensions is that their basic kernel mechanisms are not appropriate for handling computations with real-time constraints. For example, the use of fixed priorities can be a serious limitation in applications that require a dynamic creation of tasks; moreover, a single priority can be reductive to represent a task with different attributes, such as importance, deadline, period, periodicity, and so on.

There are other internal characteristics of timesharing operating systems that are inappropriate for supporting the real-time extensions. For example, most internal queues are handled with a FIFO policy, which is often preserved even in the real-time version of the system. In some system, the virtual memory management mechanism does not allow to lock pages in main memory; hence, page-fault handling may introduce large and unbounded delays on process execution. Other delays are introduced by non-preemptable system calls, by synchronous communication channels, and by the interrupt handling mechanism. These features degrade the predictability of the system and prevent any form of guarantee on the application tasks.

The observations above are sufficient to conclude that the real-time extensions of timesharing operating systems can only be used in noncritical real-time applications, where missing timing constraints does not cause serious consequences on the controlled environment.

The lack of commercial operating systems capable of efficiently handling task sets with hard timing constraints, induced researchers to investigate new computational paradigms and new scheduling strategies aimed at guaranteeing a highly predictable timing behavior. The operating systems conceived with such a novel software technology are called *hard real-time operating systems* and form the third category of systems outlined above.

The main characteristics that distinguish this new generation of operating systems include

- The ability to treat tasks with explicit timing constraints, such periods and deadlines;
- The presence of guarantee mechanisms that allow to verify in advance whether the application constraints can be met during execution;
- The possibility to characterize tasks with additional parameters, which are used to analyze the dynamic performance of the system;
- The use of specific resource access protocols that avoid priority inversion and limit the blocking time on mutually exclusive resources.

Expressive examples of operating systems that have been developed according to these principles are CHAOS [SGB87], MARS [KDK$^+$89], Spring [SR91], ARTS [TM89], RK [LKP88], TIMIX [LK88], MARUTI [LTCA89], HARTOS [KKS89], YARTOS [JSP92], and HARTIK [But93]. Most of these kernels do not represent a commercial product but are the result of considerable efforts carried out in universities and research centers.

The main differences among the kernels mentioned above concern the supporting architecture on which they have been developed, the static or dynamic approach adopted for scheduling shared resources, the types of tasks handled by the kernel, the scheduling algorithm, the type of analysis performed for verifying the schedulability of tasks, and the presence of fault-tolerance techniques.

In the rest of this chapter, some of these systems are illustrated to provide a more complete view of the techniques and methodologies that can be adopted to develop a new generation of real-time operating systems with highly predictable behavior.

11.2 MARS

MARS (MAintainable Real-time System) is a fault-tolerant distributed real-time system developed at the University of Vienna [DRSK89, KDK$^+$89] to

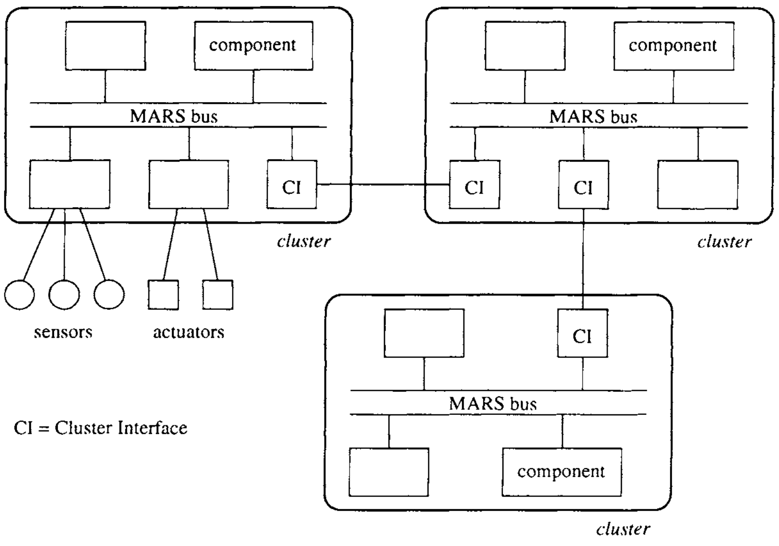

Figure 11.1 The MARS target architecture.

support complex control applications (such as air traffic control systems, railway switching systems, and so on) where hard deadlines are imposed by the controlled environment.

The MARS target architecture consists of a set of computing nodes (*clusters*) connected through high speed communication channels. Each cluster is composed of a number of acquisition and processing units (*components*) interconnected by a synchronous real-time bus, the MARS-bus. Each component is a self-contained computer on which a set of real-time application tasks and an identical copy of the MARS operating system is executed. A typical configuration of the MARS target architecture is outlined in Figure 11.1.

The main feature that distinguishes MARS from other distributed real-time systems is its deterministic behavior even in peak-load conditions; that is, when all possible events occur at their maximum specified frequency. Fault-tolerance is realized at the cluster level through active redundant components, which are grouped in a set of *Fault-Tolerant Units* (FTUs). A high error-detection coverage is achieved by the use of software mechanisms at the kernel level and hardware mechanisms at the processor level.

Within an FTU, a single redundant component fails silently; that is, it either operates correctly or does not produce any results. This feature facilitates system maintainability and extensibility, since redundant components may be removed from a running cluster, repaired, and reintegrated later, without affecting the operation of the cluster. Moreover, a component can be expanded into a new cluster that shows the same I/O behavior. In this way, a new cluster can be designed independently from the rest of the system, as long as the I/O characteristics of the interface component remain unchanged.

Predictability under peak-load situations is achieved by using a static scheduling approach combined with a time-driven dispatching policy. In MARS, the entire schedule is precomputed off-line considering the timing characteristics of the tasks, their cooperation by message exchange, as well as the protocol used to access the bus. The resulting tables produced by the off-line scheduler are then linked to the core image of each component and executed in a time-driven fashion. Dynamic scheduling is avoided by treating all critical activities as periodic tasks.

Although the static approach limits the flexibility of the system in dynamic environments, it is highly predictable and minimizes the runtime overhead for task selection. Moreover, since scheduling decisions are taken off-line, a static approach allows the use of sophisticated algorithms to solve problems (such as jitter control and fault-tolerance requirements) that are more complex than those typically handled in dynamic systems.

All MARS components have access to a common global time base, the system time, with known synchronization accuracy. It is used to test the validity of real-time information, detect timing errors, control the access to the real-time bus, and discard the redundant information.

From the hardware point of view, each MARS component is a slightly modified standard single-board computer, consisting of a Motorola 680x0 CPU, a Local Area Network Controller for Ethernet (LANCE), a Clock Synchronization Unit (CSU), two RS-232 serial interfaces, and one Small Computer System Interface (SCSI).

The software residing in a MARS component can be split into the following three classes:

1. **Operating System Kernel.** Its primary goals are resource management (CPU, memory, bus, and so on) and hardware transparency.

2. **Hard Real-Time Tasks** (HRT-tasks). HRT-tasks are periodic activities that receive, process, and send messages. Each instance of a task is characterized by a hard deadline, within which it has to be completed. The set of HRT-tasks consists of application tasks and system tasks, which perform specific functions of the kernel, such as time synchronization and protocol conversions.

3. **Soft Real-Time Tasks** (SRT-tasks). SRT-tasks are activities that are not subject to strict deadlines. Usually, they are aperiodic tasks scheduled in background, during the idle time of the processor.

All hardware details are hidden within the kernel, and all kernel data structures cannot be accessed directly. Both application tasks and system tasks access the kernel only by means of defined system calls. To facilitate porting of MARS to other hardware platforms, most of the operating system code is written in standard C language.

11.2.1 Communication

In MARS, communication among tasks, components, clusters, and peripherals occurs through a uniform message passing mechanism. All messages are sent periodically to exchange information about the state of the environment or about an internal state. State-messages are not consumed when read, so they can be read more than once by an arbitrary number of tasks. Each time a new version of a message is received, the previous version is overwritten, and the state described in the message is updated.

All MARS messages have an identical structure, consisting of a standard header, a constant length, and a standard trailer. Besides the LAN dependent standard fields, the header contains several other fields that include the observation time of the information contained in the message, the validity interval, as well as the send and receive time stamped on the message by the SCU. The trailer basically contains a checksum. The structure of the message body is defined by the application programmer, whereas its size is fixed and predefined in the system.

Since messages describe real-time entities that cannot be altered by tasks, messages are kept in read-only buffers of the operating system. Message exchange between the kernel and the application tasks does not require an explicit copy of the message, but it is performed by passing a pointer. In MARS, process

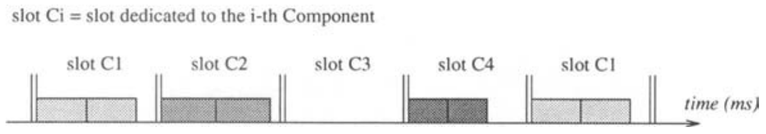

Figure 11.2 Timing of the MARS-bus using the TDMA-protocol with redundant message transmission.

communication is completely asynchronous; hence, there is no need for explicit flow control. If the sender has a frequency higher than that of the receiver, the state is updated faster than read, but no buffer overflow will occur because the latest message replaces the previous one.

Messages among components travel on the MARS-bus, which is an Ethernet link controlled by a TDMA-protocol (Time Division Multiple Access). This protocol provides a collision-free access to the Ethernet even under peak-load conditions. A disadvantage of the TDMA-protocol is a low efficiency under low-load conditions because the sending capacity of a component cannot exceed a fixed limit (approximately equal to the network capacity divided by the number of components in the cluster) even if no other component in the cluster has to send messages. Nevertheless, since MARS has mainly been designed to be predictable even under peak-load conditions, TDMA is the protocol that best satisfies this requirement. As shown in Figure 11.2, each message is sent twice on the MARS-bus.

In order to detect timing errors during communication, each message receives two time stamps from the CSU (when sent and when received), with an accuracy of about three microseconds.

11.2.2 Scheduling

In MARS, the scheduling of hard real-time activities is performed off-line considering the worst-case execution times of tasks, their interaction by message exchange, and the assignment of messages to TDMA slots. The static schedule produced by the off-line scheduler is stored in a table and loaded into each individual component. At runtime, the scheduling table is executed by a dispatcher, which performs task activation and context switches at predefined time instants. The disadvantage of this scheduling approach is that no tasks can be created dynamically, so the system is inflexible and cannot adapt to changes in the environment. On the other hand, if the assumptions on the controlled environment are valid, the static approach is quite predictable and minimizes the runtime overhead for making scheduling decisions.

Scheduling techniques that increase the flexibility of MARS in dynamic environments have been proposed by Fohler for realizing changes of operational modes [Foh93] and allowing on-line service of aperiodic tasks [Foh95].

The MARS system also allows different scheduling strategies to be adopted in different operating phases. That is, during the design phase, the programmer of the application can define several operational phases of the system characterized by different task sets, each handled by an appropriate scheduling algorithm. For example, for an aircraft control application, five phases can be distinguished: loading, taking off, flying, landing, and unloading. And each phase may require different tasks or a different scheduling policy. The change between two schedules (*mode change*) may be caused either by an explicit system call in an application task or by the reception of a message associated with a scheduling switch.

Two types of scheduling switches are supported by the kernel: a *consistent* switch and an *immediate* switch. When performing a consistent scheduling switch, tasks can only be suspended at opportune instants (determined during the design stage) so that they are guaranteed to remain in a consistent state with the environment. The immediate switch, instead, does not preserve consistency, but it guarantees that switching will be performed as soon as possible; that is, at the next invocation of the major interrupt handler, which has a period of eight milliseconds.

11.2.3 Interrupt handling

In MARS, all interrupts to the CPU are disabled, except for the clock interrupt from the CSU. Allowing each device to interrupt the CPU, in fact, would cause an unpredictable load on the system that could jeopardize the guarantee performed on the hard tasks. A priority scheme for interrupts has also been discarded because it would give advantage to high-priority devices, while low-priority devices might starve for the CPU, causing missed deadlines in consequence. Since interrupts are disabled, peripheral devices are polled periodically within the clock interrupt handler.

The clock interrupt handler is split into two sections activated with different frequencies. The first section (*minor handler*), written in assembler for efficiency

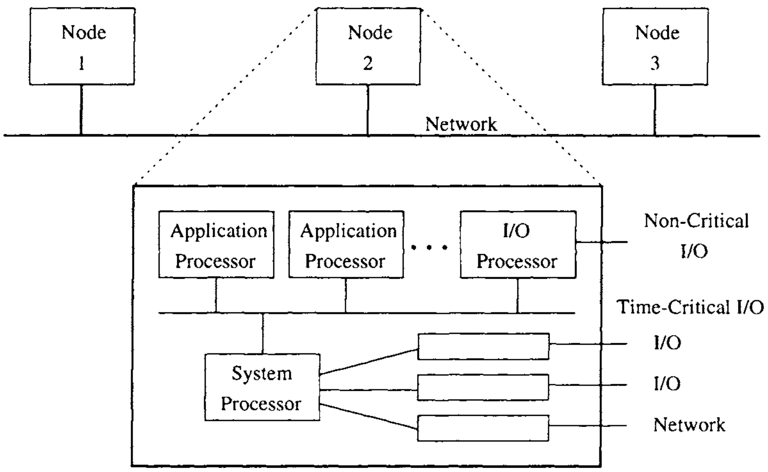

Figure 11.3 The Spring distributed architecture.

reasons, is carried out every millisecond. The second section (*major handler*), written in C, is activated every 8 milliseconds, immediately after the execution of the first part. The minor interrupt handler may suspend any system call, whereas the major handler is delayed until the end of the system call.

11.3 SPRING

Spring is a real-time distributed operating system developed at the University of Massachusetts at Amherst [SR89, SR91] for supporting large complex control applications characterized by hard timing constraints. The Spring target architecture is illustrated in Figure 11.3 and consists of a set of multiprocessor nodes connected through a high speed communication network. Each node contains three types of processors: one or more application processors, a system processor, and an I/O subsystem (front end).

- Application processors are dedicated to the execution of critical application tasks that have been guaranteed by the system.
- The system processor is responsible for executing the scheduling algorithm (a crucial part of the system) and supporting all kernel activities. Such a physical separation between system activities and application activities

allows to reduce the system overhead on the application processors and remove unpredictable delays on tasks' execution.

- The I/O subsystem is responsible for handling non-critical interrupts, coming from slow peripheral devices or from sensors that do not have a predictable response time. Time critical I/O is directly performed on the system processor.

An identical copy of the Spring kernel is executed on each application processor and on the system processor, whereas the I/O processor can be controlled by any commercial priority-based operating system. Within a node, each processing unit consists of a commercial Motorola MVME136A board, plugged in a VME bus. On this board, part of the main memory is local to the processor and is used for storing programs and private data, while another part is shared among the other processors through the VME bus.

Spring allows dynamic task activation, however the assignment of tasks to processors is done statically to improve speed and eliminate unpredictable delays. To increase efficiency at runtime, some tasks can be loaded on more processors, so that, if an overload occurs when a task is activated, the task can be executed on another processor without large overhead.

The scheduling mechanism is divided in four modules:

- At the lowest level, there is a dispatcher running on each application processor. It simply removes the next ready task from a system task table that contains all guaranteed tasks arranged in the proper order. The rest of the scheduling modules are executed on the system processor.
- The second module consists of a local scheduler (resident on the system processor), which is responsible for dynamically guaranteeing the schedulability of a task set on a particular application processor. Such a scheduler produces a system task table that is then passed to the application processor.
- The third scheduling level is a distributed scheduler that tries to find a node available in the case in which a task cannot be locally guaranteed.
- The fourth scheduling module is a metalevel controller that adapts the various parameters of the scheduling algorithm to the different load conditions.

11.3.1 Task management

In Spring, tasks are classified based on two main criteria: importance and timing requirements. The importance of a task is the value gained by the system when the task completes before its deadline. Timing requirements represent the real-time specification of a task and may range over a wide spectrum, including hard or soft deadlines, periodic or aperiodic execution, or no explicit timing constraints.

Based on importance and timing requirements, three types of tasks are defined in Spring: critical tasks, essential tasks, and unessential tasks.

- Critical tasks are those tasks that must absolutely meet their deadlines; otherwise, a catastrophic result might occur on the controlled system. Due to their criticalness, these tasks must have all resources reserved in advance and must be guaranteed off-line. Usually, in real-world applications, the number of critical tasks is relatively small compared to the total number of tasks in the system.
- Essential tasks are those tasks that are necessary to the operation of the system; however, a missed deadline does not cause catastrophic consequences, but only degrades system's performance. The number of essential tasks in typically large in complex control applications; hence, they must be handled dynamically or it would be impossible (or highly expensive) to reserve enough resources for all of them.
- Unessential tasks are processes with or without deadlines that are executed in background; that is, during the idle times of the processor. For this reason, unessential tasks do not affect the execution of critical and essential tasks. Long-range planning tasks and maintenance activities usually belong to this class.

Spring tasks are characterized by a large number of parameters. In particular, for each task, the user has to specify a worst-case execution time, a deadline, an interarrival time, a type (critical, essential, or unessential), a preemptive or non-preemptive property, an importance level, a list of resources needed, a precedence graph, a list of tasks with which the task communicates, and a list of nodes on which the task code has to be loaded. This information is used by the scheduling algorithm to find a feasible schedule.

11.3.2 Scheduling

The objective of the Spring scheduling algorithm is to dynamically guarantee the execution of newly arrived tasks in the context of the current load. The feasibility of the schedule is determined considering many issues, such as timing constraints, precedence relations, mutual exclusion on shared resources, non-preemption properties, and fault-tolerant requirements. Since this problem is NP-hard, the guarantee algorithm uses a heuristic approach to reduce the search space and find a solution in polynomial time. It starts at the root of the search tree (an empty schedule) and tries to find a leaf (a complete schedule) corresponding to a feasible schedule.

On each level of the search, a *heuristic function* H is applied to each of the tasks that remain to be scheduled. The task with the smallest value determined by the heuristic function H is selected to extend the current schedule. The heuristic function is a very flexible mechanism that allows to easily define and modify the scheduling policy of the kernel. For example, if $H = a_i$ (arrival time), the algorithm behaves as First Come First Served; if $H = C_i$ (computation time), it works as Shortest Job First; whereas if $H = d_i$ (absolute deadline), the algorithm is equivalent to Earliest Deadline First.

To consider resource constraints in the scheduling algorithm, each task τ_i has to declare a binary array of resources $R_i = [R_1(i), \dots, R_r(i)]$, where $R_k(i) = 0$ if τ_i does not use resource R_k, and $R_k(i) = 1$ if τ_i uses R_k in exclusive mode. Given a partial schedule, the algorithm determines, for each resource R_k, the earliest time the resource is available. This time is denoted as EAT_k (Earliest Available Time). Thus, the earliest start time $T_{est}(i)$ that task τ_i can begin the execution without blocking on shared resources is

$$T_{est}(i) = \max[a_i, \max_k(EAT_k)],$$

where a_i is the arrival time of τ_i. Once T_{est} is calculated for all the tasks, a possible search strategy is to select the task with the smallest value of T_{est}. Composed heuristic functions can also be used to integrate relevant information on the tasks, such as

$$\begin{aligned} H &= d + W \cdot C \\ H &= d + W \cdot T_{est}, \end{aligned}$$

where W is a weight that can be adjusted for different application environments.

Precedence constraints can be handled by introducing a new factor E, called *eligibility*. A task becomes eligible to execute only when all its ancestors in the precedence graph are completed. If a task is not eligible, it cannot be selected for extending a partial schedule.

While extending a partial schedule, the algorithm determines whether the current schedule is *strongly feasible*; that is, it is also feasible by extending it with any of the remaining tasks. If a partial schedule is found not to be strongly feasible, the algorithm stops the search process and announces that the task set is not schedulable; otherwise, the search continues until a complete feasible schedule is met. Since a feasible schedule is reached through n nodes and each partial schedule requires the evaluation of at most n heuristic functions, the complexity of the Spring algorithm is $O(n^2)$.

Backtracking can be used to continue the search after a failure. In this case, the algorithm returns to the previous partial schedule and extends it by the task with the second-smallest heuristic value. To restrict the overhead of backtracking, however, the maximum number of possible backtracks must be limited. Another method to reduce the complexity is to restrict the number of evaluations of the heuristic function. Do to that, if a partial schedule is found to be strongly feasible, the heuristic function is applied not to all the remaining tasks but only to the k remaining tasks with the earliest deadlines. Given that only k tasks are considered at each step, the complexity becomes $O(kn)$. If the value of k is constant (and small, compared to the task set size), then the complexity becomes linearly proportional to the number of tasks.

11.3.3 I/O and interrupt handling

In Spring, peripheral I/O devices are divided in two classes: slow and fast I/O devices. Slow I/O devices are multiplexed through a front-end dedicated processor (I/O processor), controlled by a commercial operating system. Device drivers running on this processor are not subject to the dynamic guarantee algorithm, although they can activate critical or essential tasks. Fast I/O devices are handled by the system processor, so they do not affect the execution of application tasks. Interrupts from fast I/O devices are treated as instantiating a new task that is subject to the guarantee routine just like any other task in the system.

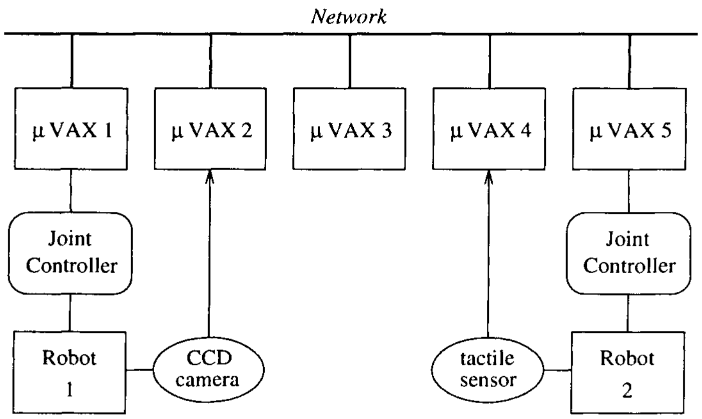

Figure 11.4 Target architecture for RK.

11.4 RK

RK (Real-time Kernel) is a distributed real-time system developed at the University of Pennsylvania [LKP88, LK88] to support multisensor robotic applications. The presence of hard timing constraints in robotic control activities is necessary for two important reasons. First, sensors and actuators require regular acquisition and feedback control in order to achieve continuous and smooth operations. Second, some high-level tasks (such as trajectory planning, obstacle avoidance, and so on) may require timely execution to avoid possible catastrophic results.

The target architecture for which RK has been designed is illustrated in Figure 11.4. It consists of five processors (MicroVAX) connected through a 10 Mb Ethernet, two robot manipulators (PUMA 560) with a joint controller each, a tactile sensor, and a camera. One of the processors (P3) works as a supervisor, two (P1 and P5) are connected to the joint controllers via a parallel interface, one (P2) is responsible for image acquisition and processing, and one (P4) is dedicated to the tactile sensor. In order to support all sensory and control activities needed for this robot system, an identical copy of the kernel is executed on each of the five processors.

To achieve predictable behavior, RK provides a set of services whose worst-case execution time is bounded. In addition, the kernel allows the programmer to specify timing constraints for process execution and interprocess communication.

11.4.1 Scheduling

RK supports both real-time and non-real-time tasks. Real-time tasks are divided in three classes with different level of criticalness: imperative, hard, and soft. The assignment of the CPU to tasks is done according to a priority order. Within the same class, imperative processes are executed on a First-Come-First-Served (FCFS) basis, whereas hard and soft processes are executed based on their timing constraints by the EDF algorithm. The difference between hard and soft tasks is that hard tasks are subject to a guarantee algorithm that verifies their schedulability at creation time, whereas soft tasks are not guaranteed. Finally, non-real-time tasks are scheduled in background using FCFS. Timing constraints on real-time tasks can also be specified as periodic or sporadic and can be defined on the whole process, on a part of a process, and on messages.

To facilitate the programming of timing constraints, RK supports a notion of *temporal scope*, which identifies explicit timing constraints with a sequence of statements. Each temporal scope consists of five attributes: a hard/soft flag, a start time, a maximum execution time, a deadline, and a unique identifier. Whenever a temporal scope with a hard flag is entered, the scheduler checks whether the corresponding timing constraints can be guaranteed in the context of the current load. If the request cannot be guaranteed, an error message is generated by the kernel.

A timing constraint is violated if either a process executes longer than the maximum declared execution time or its deadline is exceeded. When this happens, the kernel sends a signal to the process. If the process is hard, a critical system error has occurred (since the timing constraint was guaranteed by the scheduler); thus, the task that missed the deadline becomes an *imperative* process, and a controlled shutdown of the system is performed as soon as possible.

11.4.2 Communication

RK provides three basic communication methods among real-time tasks:

- Signals, for notification of critical system errors;
- Timed events, for notification of events with timing constraints;
- Ports, for asynchronous message passing with timing constraints.

Signals

Signals are used by the kernel to notify that an error has occurred. The purpose of sending such a signal is to give the process a chance to clean up its state or to perform a controlled shutdown of the system. There are three types of errors: timing errors, process errors, and system errors. Timing errors occur when either a process executes longer than its maximum execution time or its deadline is exceeded. Process errors occur when a task executes an illegal operation – for example, an access to an invalid memory address. System errors are due to the kernel; for example, running out of buffers that have been guaranteed to a task. When the kernel sends a signal to a process, the process executes an appropriate signal handler and then resumes the previous execution flow when the handler is finished.

Timed events

Events are the most basic mechanism for interprocess communication. Unlike a signal, an event can be sent, waited on, delayed, and preempted. In addition, each event can have timing constraints and an integer value, which can be used to pass a small amount of data. For each event, the kernel remembers only the last occurrence of the event. Thus, if an event arrives while another one of the same type is pending, only the value of the last one is remembered.

Like signals, whenever a process receives an event, it executes an associated event handler; the previous execution flow resumes once the handler is finished. There are two ways to associate timing constraints with events. According to the first way, the receiver of an event may specify a timeout for executing the event handler. Alternatively, the sender may include a deadline when the event is sent. If both the sender and the receiver specify timing constraints for the same event, then the earliest deadline is used for the execution of the handler.

If a non-real-time process receives a timed event, the corresponding event handler is executed immediately, and, during the handling of the event, the process is treated as real-time. This feature allows non-real-time server processes to handle requests from real-time processes.

Ports

The port construct is widely used in operating systems for interprocess communication. In RK, it is extended for real-time communication by allowing the sender to specify timing constraints in messages and the receiver to control message queueing and reception strategies. Sending a message to a port is always nonblocking, and the execution time for a transmission is bounded to ensure a predictable delay. For critical message communication, the sender can include a set of timing attributes within each message, such as the start time, the maximum duration and the deadline. Receiving a message can be either explicit or asynchronous. When using an explicit receive primitive, the process can specify a timeout to limit the delay in waiting for a message. For asynchronous receive, the receiver associates a timed event with a port and each message arrival is notified through the timed event.

Every RK process is created with a default reception port, used during initialization and to request services from system server processes. Additional ports can be created using the following system call:

port_id = **port_create**(type).

The argument *type* specifies whether the port is for receiving messages or for multicasting messages. For a reception port, any process can send a message to it, but only the creator can receive from it. A multicast port realizes a *one-to-many* communication channel. Each multicast port has a list of destination ports to which messages are to be forwarded. When a message is sent to a multicast port, it is forwarded to all ports connected to it, and this forwarding is repeated until the message reaches a reception port.

When creating a reception port, various attributes can be specified by the creator. They allow the following characteristics to be defined:

- **The ordering of messages within the port queue**. It can be done either by transmission time, arrival time, or deadline.
- **The size of the queue** – that is, the maximum number of messages that can be stored in the queue. In case of overflow, it is possible to specify whether messages are thrown away at the head or at the tail of the queue.
- **Communication semantics**. Normally, messages are removed from the queue when they are received. However, when the *stick* attribute is set, a message remains in the queue even after it is received, and it is replaced only when a new message arrives.

In RK, the *send* and *receive* system calls have the following syntax:

send(portid, reply_portid, t_record, msg, size);

receive(portid, reply_portid, timeuot, t_record, msg, size);

where *portid* is the identifier of the port; *reply_portid* specifies where to send a reply; *timeout* (only in reception) specifies the maximum amount of time that the primitive should block waiting for a message; *t_record* is a pointer to a record containing the three timing attributes (start time, maximum duration, and deadline) specified by the sender; *msg* is a pointer to the message; and *size* is the size of the message.

11.4.3 I/O and interrupt handling

Traditional operating systems provide device drivers that simplify the interactions between application processes and peripheral devices. This approach allows the same device to be used by many processes; however, it introduces additional delays during processes's execution that may jeopardize the guarantee of hard real-time activities.

In robotics applications, this problem is not so relevant, since sensory devices are not shared among processes but are controlled by dedicated tasks that collect data and preprocess them. For this reason, RK allows processes to directly control devices by sharing memory and accessing device registers. In addition, a process may request the kernel to convert device interrupts into timed events.

Although this approach requires the programmer to know low-level details about devices, it is faster than the traditional method, since no context switching is needed to apply feedback to a device. Furthermore, the kernel needs not to be changed when removing or adding new devices.

11.5 ARTS

ARTS (Advanced Real-time Technology System) is a distributed real-time operating system developed at the Carnegie Mellon University [TK88, TM89] for verifying advanced computing technologies for a distributed environment. The target architecture for which ARTS has been developed consists of a set of SUN workstations connected by a real-time network based on IEEE 802.5 To-

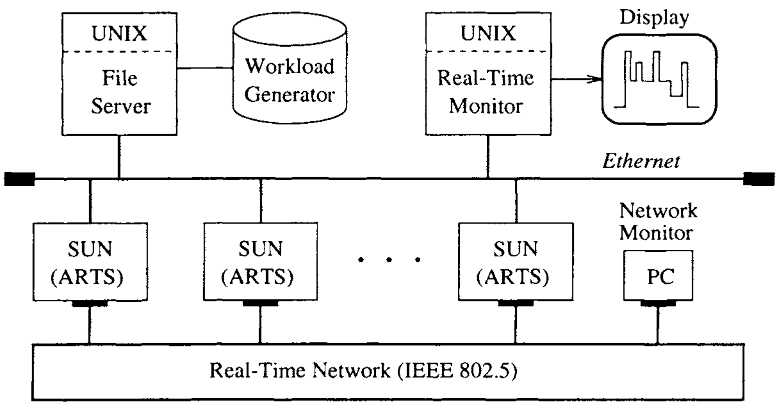

Figure 11.5 The ARTS target architecture.

ken Ring. Figure 11.5 shows the typical configuration of the system and the relation between the kernel and its real-time tools.

The programming environment provided by the ARTS system is based on an object-oriented paradigm, in which every computational entity is represented by an object. Objects can be defined as real-time or non-real-time objects. Each operation associated with a real-time object has a worst-case execution time, called a *time fence*, and a time exception handling routine. In addition, an ARTS object can be *passive* or *active*. Active objects are characterized by the presence of one or more internal threads (defined by the user) that accept incoming invocation requests.

All threads are implemented as lightweight processes that share the same address space. A thread can be defined as a periodic or aperiodic task depending on its timing attributes. The timing attributes of a thread consist of a value function, a worst-case execution time, a period, a phase, and a delay value.

ARTS supports the creation and destruction of objects at a local node, as well as at a remote node. Although process migration is a very important mechanism in non-real-time distributed operating systems, the ARTS kernel does not support object migration during runtime. Instead, it can move an object by shutting down the activities and reinitiating the object at the target host with appropriate parameters.

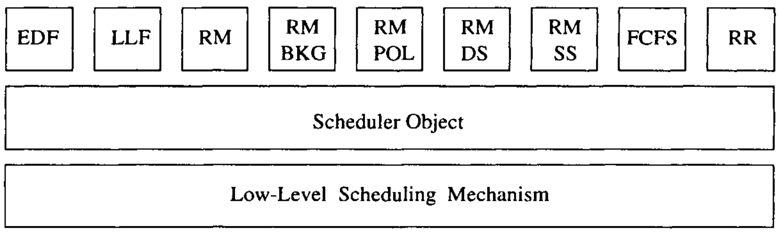

Figure 11.6 Structure of the ARTS scheduler.

11.5.1 Scheduling

In ARTS, the scheduling policy is implemented as a self-contained kernel object and is separated from the thread handling mechanism, which performs only dispatching and blocking.

For experimental purposes, several scheduling policies have been implemented in the ARTS kernel, including static algorithms such as Rate Monotonic (RM) and dynamic algorithms such as Earliest Deadline First (EDF) and Least Laxity First (LLF). In conjunction with Rate Monotonic, a number of strategies for handling aperiodic threads have been realized, such as Background servicing (BKG), Polling (POL), Deferrable Server (DS), and Sporadic Server (SS). More common scheduling algorithms such as First Come First Served (FCFS) and Round Robin (RR) have also been realized for comparison with real-time scheduling policies. A scheduling policy object can be selected either during system initialization or during runtime. Figure 11.6 shows the general structure of the ARTS scheduler.

A schedulability analyzer associated with each scheduling algorithm allows the following to be guaranteed:

- The feasibility of hard tasks within their deadlines,
- A high cumulative value for soft tasks, and
- Overload control based on the value functions of aperiodic tasks.

When selecting a server mechanism for handling aperiodic tasks, the server parameters (period and capacity) are set to fully utilize the processor. This allows to reserve the maximum CPU time for aperiodic service while guaranteeing the schedulability of periodic hard tasks.

11.5.2 Communication

In traditional real-time operating systems, interprocess communication mechanisms are realized to be fast and efficient (that is, characterized by a low overhead). In ARTS, however, the main goal has been to realize a communication mechanism characterized by a predictable and analyzable behavior. To achieve this goal, ARTS system calls require detailed information about communication patterns among objects, including the specification of periodic message traffic and rates for aperiodic traffic.

In ARTS, every message communication is caused by an invocation of a target object's operation, and the actual message communication is performed in a Request-Accept-Reply fashion. Unlike traditional message passing paradigms, the caller must specify the destination object, the identifier of the requested operation, the pointer to the message, and the pointer to a buffer area for the reply message.

To avoid priority inversion among objects inside each node, message transmission is integrated with a Priority Inheritance mechanism, which allows to propagate priority information across object invocations. All network messages are handled by a Communication Manager (CM), where different protocols are implemented using a state table specification. The CM prevents priority inversion over the network by using priority queues with priority inheritance. Thus, if a low-priority message is processed when a higher-priority message arrives, the low-priority message will execute at the highest priority. In this way, the highest-priority message remains in the queue for at most the time it takes to process one message.

11.5.3 Supporting tools

ARTS provides a set of supporting tools, the ARTS Tool-Set [TK88], aimed at reducing the complexity of application development in a distributed real-time environment. This tool-set includes a schedulability analyzer, a support for debugging, and a system monitoring tool.

Schedulability analyzer

The main objective of this tool is to verify the schedulability of a given set of hard real-time tasks under a particular scheduling algorithm. The performance of soft aperiodic tasks are computed under specific service mechanisms, such as

Background, Polling, Deferrable Server, and Sporadic Server. An interactive graphical user interface is provided on a window system to quickly select the scheduling algorithm and the task set to be analyzed. To confirm the schedulability of the given task set in a practical environment, this tool also includes a synthetic workload generator, which creates a particular sequence of requests based on a workload table specified by the user. The synthetic task set can then be executed by a scheduling simulator to test the observance of the hard timing constraints.

Debugging

The ARTS system provides the programmer with a set of basic primitives that can be used for building a debugger and for monitoring process variables. For example, the *Thread_Freeze* primitive halts a specific thread for inspection, while the *Object_Freeze* primitive stops the execution of an ARTS object (that is, all its associated threads). *Thread_Unfreeze* and *Object_Unfreeze* primitives resume a suspended thread and object, respectively. While a thread is in a *frozen* state, the *Fetch* primitive allows to inspect its status in terms of a set of values of data objects. The value of any data object can be replaced using the *Store* primitive. Finally, the *Thread_Capture* and *Object_Capture* primitives allow to capture on-going communication messages from a specified thread and object, respectively.

System monitoring

ARTS includes a monitoring tool, called *Advance Real-time Monitor* (ARM), whose objective is to observe and visualize the system's runtime behavior. Typical events that can be visualized by this tool are context switches among tasks caused by scheduling decisions. ARM is divided into three functional units: the *Event Tap*, the *Reporter*, and the *Visualizer*. The Event Tap is a probe embedded inside the kernel to pick up the row data on interesting events. The Reporter is in charge of sending the row data to the Visualizer on a remote host, which analyzes the events and visualizes them in an interactive graphical environment. The Visualizer is designed to be easily ported to different graphical interfaces.

11.6 HARTIK

HARTIK (HArd Real-TIme Kernel) is a hard real-time operating environment developed at the Scuola Superiore S. Anna of Pisa [BDN93, But93] to support advanced robot control applications characterized by stringent timing constraints.

Complex robot systems are usually equipped with different types of sensors and actuators and hence require the concurrent execution of computational activities characterized by different types of timing constraints. For example, processing activities related to sensory acquisition and low-level servoing must be periodically executed with regular activation rates to ensure a correct reconstruction of external signals and guarantee a smooth and stable behavior of the robot system. Other activities (such as planning special actions, modifying the control parameters, or handling exceptional situations) are intrinsically aperiodic and are triggered when some particular condition occurs. To achieve a predictable timing behavior and to satisfy system stability requirements, most acquisition and control tasks require stringent timing constraints, that have to be met in all anticipated workload conditions. In addition, complex robot systems are typically built using disparate peripheral devices that may be distributed on heterogeneous computers.

For the reasons mentioned above, HARTIK has been designed to support the following major characteristics:

- **Flexibility**. It is possible to schedule hybrid task sets consisting of periodic and aperiodic tasks with different level of criticalness.
- **Portability**. The kernel has been designed in a modular fashion, and all hardware-dependent code is encapsulated in a small layer that provides a virtual machine environment.
- **Dynamic preemptive scheduling and on-line guarantee**. Any hard task is subject to a feasibility test. If a task cannot be guaranteed, the system raises an exception that allows to take an alternative action.
- **Efficient aperiodic service**. An integrated scheduling algorithm enhances responsiveness of soft aperiodic requests without jeopardizing the guarantee of the hard tasks.
- **Predictable resource sharing**. Special semaphores allow to bound the maximum blocking time on critical sections, preventing deadlock and chained blocking.

- **Fully asynchronous communication.** A particular nonblocking mechanism, called CAB, is provided for exchanging messages among periodic tasks with different periods, thus allowing the implementation of multilevel feedback control loops.

- **Efficient and predictable interrupt handling mechanism.** Any interrupt request can either be served immediately, or cause the activation of an handler task, which is guaranteed and scheduled as any other hard task in the system.

To facilitate the development of real-time control applications on heterogeneous architectures, HARTIK has been designed to be easily ported on different hardware platforms. At present, the kernel is available for Motorola MC 680x0 boards with VME bus, Intel 80x86 and Pentium with ISA/PCI bus, and DEC AXP-Alpha stations with PCI bus.

Figure 11.7 illustrates a possible architecture that can be used to build a control application. In this solution, control algorithms, trajectory planning, and feedback loops are executed on a Pentium-based computer; sensory acquisition and data preprocessing are executed on a Motorola 68030 processor; whereas the application development is carried out on a DEC Alpha workstation. In this node, a set of tools is available for designing the application structure, estimating the maximum execution time of the tasks, analyzing the schedulability of the task set, and monitoring the system activity.

11.6.1 Task management and scheduling

HARTIK distinguishes three classes of tasks with different criticalness:

- **HARD tasks.** They are periodic or aperiodic processes with critical deadline that are guaranteed by the kernel at creation time. Moreover, the system performs a runtime check on hard deadlines, notifying a time overflow when a hard deadline is missed.

- **SOFT tasks.** They are periodic or aperiodic processes with non-critical deadline that are not guaranteed by the system. Soft tasks are handled by the Total Bandwidth Server[SB94, SB96], which enhances their response time without jeopardizing the guarantee of hard tasks.

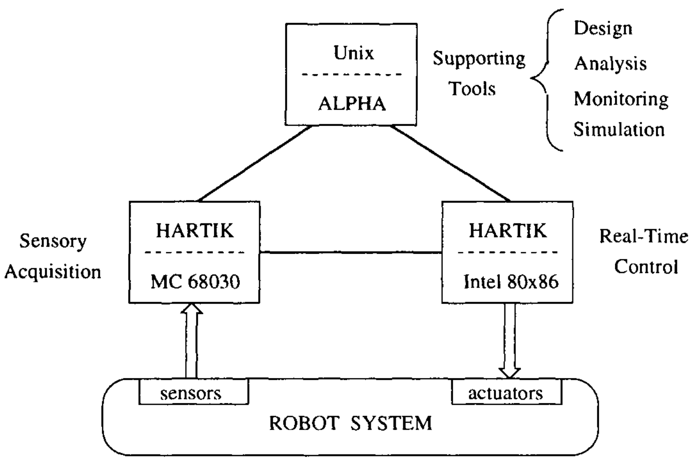

Figure 11.7 Example of heterogeneous architecture that can be used with HARTIK.

- **NRT tasks.** They are Non-Real-Time aperiodic processes with no timing constraints. NRT tasks are scheduled in background and are characterized by a static priority level assigned by the user.

When a task is created, several parameters have to be specified, such as its name, its class (HARD, SOFT, or NRT), its type (periodic or aperiodic), a relative deadline, a period (or a minimum interarrival time for sporadic tasks), a worst-case execution time, a pointer to a list of resources handled by the Stack Resource Policy, and a maximum blocking time. Hard and soft tasks are scheduled according to the Earliest-Deadline-First scheduling policy, which is optimal and achieves full processor utilization.

Real-time tasks can share resources in a predictable fashion through the *Stack Resource Policy (SRP)*. The SRP ensures that, once started, a task will never block until completion but can be preempted only by higher-priority tasks. Furthermore, the SRP avoids priority inversion, chained blocking, deadlock, and reduces the number of context switches due to resource acquisition. Using SRP, the maximum blocking time that any task can experience is equal to the duration of the longest critical section, among those that can block it.

11.6.2 Process communication

HARTIK provides both synchronous and asynchronous communication primitives to adapt to different task requirements. For synchronous communication, tasks can use two types of ports: RECEIVE and BROADCAST.

A RECEIVE port is a channel where many tasks can send messages to, but only one, the owner, is allowed to receive them. Sending messages to and receiving messages from a receive port is always synchronous with timeout. Hence, these ports can be used by soft and NRT tasks and by those hard tasks that must absolutely perform synchronous communication.

BROADCAST ports provide a one-to-many communication channel. They have not only some buffering capability for incoming messages but also a list of destination ports to which messages are to be forwarded. When a message is sent to a broadcast port, it is redirected to all ports specified in the list. BROADCAST ports allow asynchronous send, but they are not directly addressable by a receive. These ports are suited for soft and non-real-time tasks.

A third type of port available in the kernel is the STICK port, which is a *one-to-many* communication channel with asynchronous semantics. When a process receives a message from a STICK port, the port does not consume the message but leaves it stuck until it is overwritten by another incoming message. As a consequence, a process is never blocked for an empty or full buffer. For this property, the use of STICK ports is strongly recommended for exchanging state information among HARD tasks.

Asynchronous communication is supported by the *Cyclic Asynchronous Buffer* (CAB) mechanism, purposely designed for the cooperation among periodic activities with different activation rate, such as sensory acquisition and control loops. A CAB provides a one-to-many communication channel which contains, at any instant, the latest message inserted in its structure.

A message is not consumed by a receiving task, but it is maintained into the CAB until a new message is overwritten. In this way, a receiving task will always find data in a CAB, so that unpredictable delays due to synchronization can be eliminated. It is important to point out that CABs do not use semaphores to protect their internal data structures, so they are not subject to priority inversion.

CAB messages are always accessed through a pointer, so that the overhead of CAB primitives is small and independent of the message size. The kernel also allows tasks to perform simultaneous read and write operations to a CAB. This is achieved through the use of multiple memory buffers. For example, if a task wants to write a new message in a CAB that is being used by another task (which is reading the current message), a new buffer is assigned to the writer, so that no memory conflict occurs. As the writing operation is completed, the written message becomes the most recent information in that CAB, and it will be available to any other task. The maximum number of buffers needed for a CAB to avoid blocking must be equal to the number of tasks that share the CAB plus one.

11.6.3 Interrupt handling

In HARTIK, a device driver is split into two parts: a *fast handler* and a *safe handler*. When an interrupt is triggered by an I/O device, the fast handler is executed in the context of the currently running task to avoid the overhead due to a context switch. It typically performs some basic input/output operations and acknowledges the peripheral. Then, the kernel automatically activates the safe handler, which is subject to the scheduling algorithm as any other aperiodic task in the system. The safe handler can be declared as a soft or sporadic task depending on the characteristics of the device. It is in charge of doing any remaining computation on the device – for example, data multiplexing among user tasks. This approach is quite flexible, since it allows to nicely combine two different service techniques: the event-driven approach (obtained by the fast handler) and the time-driven approach (obtained by the safe handler).

11.6.4 Programming tools

The HARTIK system includes a set of tools [ABDNB96, ABDNS96] to assist the development of time-critical applications from the design stage to the monitoring phase. In particular, the tool set includes a design tool to describe the structure of the application, a schedulability analyzer to verify the feasibility of critical tasks, a scheduling simulator to test the performance of the system under a synthetic workload, a worst-case execution time estimator, and a tracer to monitor and visualize the actual evolution of the application.

Design tool

The design tool includes an interactive graphics environment that allows the user to describe the application requirements according to three hierarchical levels. At the highest level, the application is described as a number of virtual nodes that communicate through channels. Virtual nodes and channels are graphically represented by icons linked with arrows. Opening the icon of a virtual node we reach the second hierarchical level. At this stage, the developer specifies the set of concurrent tasks running in the virtual node and communicating through shared critical sections or through channels. Tasks, shared resources, and channels are graphically represented by icons that the developer can move and link with arrows. Any possible object (a task, a resource, a channel, or a message) is an instance of a class for that type of object.

Scheduling analyzer

The Schedulability Analyzer Tool (SAT) is very useful for designing predictable real-time applications because it enables the developer to analyze a set of critical tasks and statically verify their schedulability. If the schedulability analysis gives a negative result, the user can change the task parameters and rerun the guarantee test. For instance, some adjustments are possible by rearranging the task deadlines or by producing a more compact and efficient code for some critical tasks or even changing the target machine.

Scheduling simulator

Many practical real-time applications do not contain critical activities but only tasks with soft time constraints, where a deadline miss does not cause any serious damage. In these applications, the user may be interested in evaluating the performance of the system in the average-case behavior rather than in the worst-case behavior. In order to do that, a statistical analysis through a graphic simulation is required. For this purpose, the tool kit includes a scheduling simulator and a load generator for creating random aperiodic activities. Actual computation times, arrival times, durations, and positions of critical sections in the tasks are computed by the load generator as random variables, whose distribution is provided by the user.

Maximum execution time evaluator

The execution time of tasks it is estimated by a proper tool, which performs a static analysis of the application code, supported by a programming style and specific language constructs to get analyzable programs. The language used to develop time-bounded code is an extension of the C language, where monitors are added to isolate and evaluate the duration of critical sections. Optional bounds are programmable to limit the number of iterations in loop statements or to limit the maximum number of processing conditional branches inside loops. The present implementation has models of Intel i386 and i486 CPUs, but the tool can be easily adapted to different kind of processors.

The model includes the simulation of the processor in a table-driven fashion, where assembly instructions are translated into execution times depending on their operating code, operands, and addressing mode. The tool works in conjunction with the C compiler and produces a graph representation of the program's control structure in terms of temporal behavior, where a weight is assigned to every branch of the graph, corresponding to the number of CPU cycles needed for the execution of a segment of sequential code. With this representation, calculating the worst-case behavior of an algorithm means evaluating the maximum cost path in the graph.

Real-time tracer

This tool allows the monitoring of the system evolution while an application is running. It consists of four main parts: a probe, a data structure in the kernel, an event recorder, and a visualizer. The probe is a kernel routine inserted in the system calls, capable of keeping track of all events occurring in the system. At each context switch, the probe saves in main memory the system time (with a microsecond a resolution) at which the event takes place, the name of the recorded primitive, the process identifier, its current deadline, and its state before the primitive execution. At system termination, the recorder saves the application trace in a file, which can be later interpreted and displayed by the visualizer. This tool produces a graphics representation of the system evolution in a desired time scale, under Windows NT/95.

The user has the possibility of moving along the trace, changing the scale factor (zoom), and displaying information about task properties, such as type, periodicity class, deadline, and period. Statistical information on waiting times into the various queues are also calculated and displayed both in graphical and textual fashion. On-line help is also provided.