The DUNE_iX Real-Time Operating System

Jean-Serge Banino Lectra Systems;
Jean Delcoigne
CEA/LETI/DEIN, Centre d'Etudes Nucléaires de Saclay;
Claude Kaiser Laboratoire Cedric, Conservatoire National des Arts et Métiers;
and Gérard Morisset Lectra Systems

ABSTRACT: Real-time applications need guarantee of response deadline by the computing system, promptness of reflex responses, reliability of application code.

The first part of the paper examines the requirements for real-time operating systems and ends with the Dune_iX basic design decisions.

The operating system must be able to provide immediate tasks for reflex reaction to interrupts and autonomous tasks for reacting with a specified deadline to periodic or aperiodic events related to the real-time application. Thus the behaviour of the computing system must be thoroughly controlled.

The guarantee of response is provided by a high level real-time scheduler which supervises the priorities given to the tasks, and by a priority-driven, reentrant, preemptive, real-time kernel.

In order to provide fast response results, the real-time kernel takes advantage of the symmetric multiprocessor architecture.

The reliability of the application code is eased by providing programming tools and allowing code reusability through full Unix compatibility.

The second part presents the detailed implementation choices of Dune_iX which aim at reducing all known kinds of latencies due to processors, resources or I/Os contentions. Immediate task association with an interrupt level and with an autonomous application task context, inter-task shared memory segments, priority inheritance, deadlock prevention, contiguous files are some of the relevant features which are provided.

The third part rapidly describes an example of a host architecture for this operating system and the last part gives some performance measures on this host architecture.

This paper presents the design requirements and the technical solutions that led to the Dune_iX Real-Time Operating System.

Dune_iX, released in 1991, corresponds to a recent evolution of Real-Time Operating System.

Major aspects of Dune_iX design are:

- true deterministic real-time behaviour,
- full compatibility with standard Unix interface,
- symmetrical multiprocessor implementation.

1 Real-Time Requirements

1.1 What is Real-Time Computing?

Real-time applications include a large spectrum of applications, such as embedded systems, process control, mobile control, nuclear power plants, laboratory experiments, robotics, and even banking systems.

Application dependent events occur periodically or randomly and force the computing system to react within a fixed time delay or at a

given date, and within a small time window, in order to capture external data, to initiate or terminate some activities, to send responses or commands.

A real-time computing system may be designed:

- as a snapshot generator system which triggers tasks fast enough to periodically catch the external world behaviour through sampling,
- or as a reactive system including tasks which answer to external prompts,
- or as a mix of both, scheduling the execution of periodic and aperiodic tasks.

Application requirements lead to a distinction between soft real-time and hard real-time. Applications present hard real-time requirements when the failure to meet timing constraints (tasks or message deadlines, sampling dates, timing dispersions within a set of "simultaneous" measurements . . .) will result into economical, human or ecological disasters.

Real-time is a serious problem for operating systems and is often misunderstood. Real-time systems, no matter how large they are, are first characterized by time constraints. The correctness of the reaction depends not only upon the logical result of a computation, but also upon the time at which the result is available by the application. "The right answer late is still wrong." [Bennet 1988, Small 1988, Burns 1990, Burns 1991, Levi 1990]

Common Misconceptions

Some authors such as G. Lelann [Lelann 1990] and J. Stankovic [Stankovic 1988] have listed several common misconceptions. Let us recall three of them, which we consider the most penalizing.

a) *"Real-time is fast computing; thus advances in supercomputer hardware, in bus or network traffic bandwidth, in computation algorithms, will take care of real-time requirements".*

Fast computing will reduce the average execution time of a given set of tasks. However the problem is to meet the individual time requirements of each reaction. Fast computing is not enough, the most important property that has to be guaranteed is response time predictability.

b) *"Real-time is fast priority handling and fast context switching; thus preemption latencies will be reduced and the tasks will start as soon as possible".*

Rapidity is not the sole issue; a fast but too frequent switching might be penalizing. Usual scheduling policies are often dumb, in order to take fast scheduling decisions. The problem is to take the right decisions based upon time constraints; of course once the decision has been taken, the execution has to be fast and must occur at the right moment. There is an optimum time window when landing a jet plane: landing too early would cause the plane to crash, landing too late would run the plane out of the strip. A strict integer based fixed priority cannot adequately describe time constraints, model the time laxity and also cope with random events. Dynamic priorities and preemption are compulsory and appropriate deterministic scheduling is often necessary. At least, it is required that once the most urgent task has been determined to run "hic et nunc", no service with lower urgency will partly or fully preempt the resources needed by this most urgent task.

c) *"Real-time is assembly coding, priority interrupt programming and device driver writing; high level languages and complex programming tools lead to unefficient code; assembly coding allows to use machine level optimization techniques".*

Low level optimization produces only low-level benefits; better efficiency results from global optimization or from carefully chosen hardware improvements. Real-time applications need to be highly reliable; their code has a significant size (several hundred thousand of machine instructions); they have a long lifetime (several decades) and will have to accept several hardware changes. Reliance on clever hand-coding and difficult to trace timing assumptions is a major source of bugs and clearly a very poor engineering management technique when one seeks for quality.

1.2 Real-Time Operating System Facilities.

A modern real-time operating system should provide facilities to fulfill the three major requirements of real-time applications. These are:

- guarantee of response from the computing system,
- promptness of a response, once it has been decided,
- reliability of the application code.

1.2.1 Time Driven Scheduling

Operating systems decisions are taken according to scheduling policies. [Habermann 1976, Lister 1984, Tanenbaum 1987]

It is important to realize that scheduling problems in real-time systems are different from the scheduling problem usually considered in classical operating systems.

a-*Usual operating system scheduling*

In usual operating systems, CPU activity is optimized to provide maximum throughput with the constraint of favouring some class of tasks. The primary concern is resource utilization instead of time constraints. Long term scheduling policies activate tasks according to application requirements (payroll listing every month, . . .) and to resource availability (mounted disks for payroll files, . . .), while short term scheduling optimizes resource utilization. In interactive systems (sometimes called time-sharing systems), complex scheduling policies enforce fairness among on-line users and favour short jobs.

In usual operating systems, all tasks are considered as aperiodic with unknown date of arrival and unknown duration. They have no compulsory execution deadlines.

b-*Real-time systems constraints*

A real-time operating system must be able to take into account periodic tasks (as a reaction to periodic events) with fixed period and fixed deadlines, as well as with aperiodic sporadic tasks (events) with unknown date of occurrence but with fixed deadlines. In both cases the duration of a task execution is not completely known since it can depend on the current values of data, on input-output transfer delays, on shared resource availability, on fault tolerance. Time constraints not solely apply to individual tasks but often to collections of cooperating tasks which are activated to react to events. This cooperation may add precedence relations and resource conflicts which lead to face an end-to-end timing analysis.

Whilst external events, resource conflicts, task duration and fault occurrences cannot completly be characterized and remain partly random, the system must be controlled such that its timing behaviour is understandable, bounded and predictable.

Thus, the determinacy requires that the user timing specifications

of the computing system behaviour are taken into account at a primary decision level and are used to control the system, whilst all operating system latencies must be precisely bounded.

These properties can be aimed at by a layered approach based on a real-time task scheduler and on a real-time kernel.

b1-*The real-time task scheduling*

Given a priori static acceptance test for schedulability of a set of periodic tasks with given period, execution time and deadlines, the real-time task scheduler can use heuristics [Chetto 1989] to dynamically estimate the processor load needed to fulfill the periodic tasks deadline constraints. The remaining processing activity, called the system laxity, is usable to cope with random sharing of the resources and with sporadic arrival of tasks. If additional processing power has to be preserved for the sporadic events, some periodic tasks may be suppressed or replaced by simpler ones. Depending on application requirements, penalty functions may be defined according to the number of important tasks that miss their deadlines; a suppression rank may be dynamically evaluated to control the abortion of tasks and to save enough CPU time for facing avalanches of urgent events which are produced by catastrophic situations.

A hard deadline scheduler can be added, assigning static priorities to periodic tasks (rate monotonic scheduling) or using dynamic priorities and preemptive scheduling (earliest deadline or least laxity).

The real-time task scheduler is in charge of determinacy in the large, and cares with the user defined constraints on all the tasks of the application.

b2-*The short time latencies*

The operating system kernel must enforce the real-time behaviour assumed by the real-time task scheduler, i.e. promptness and known latency. The timing predictions must include the insurance that the resources are available on time and therefore cope with access conflicts and fault tolerance.

The real-time kernel is in charge of determinacy in the small and cares with the constraints of individual tasks.

1.2.2 Real-Time Kernel

A real-time system can be viewed as a three-stage pipeline: data acquisition from sensors, data processing and output to activators or displays.

The real-time kernel must provide an efficient mechanism for all these stages (efficiency meaning here no latency, good choice, simplicity, rapidity . . .).

a-I/O management and control

For data acquisition and for activators, it must provide extensive I/O capabilities, such as:

- a fast and flexible input and output processing power in order to rapidly capture the data associated with the priority events, or to promptly supply the actuators or the displays,
- the absence of I/O latency caused by file granularity and by I/O buffer management, and therefore the capability of predicting the transfer delays of prioritised I/O.

b-Task management and control

For data processing, the real-time kernel must provide mechanisms which really fulfill the timing requirements:

- concurrency between kernel calls, limited only by the mutual exclusion to sensitive data, i.e. a fully preemptive and reentrant kernel,
- fast and efficient synchronisation primitives which will avoid unnecessary context switching,
- swift task context switch,
- an accurate granularity of time servers,
- a task scheduling which respects the user defined priority, and which does not cause unexpected task switching nor priority inversion [Kaiser 1983].

c-Resource management and control

In a multitasking system there will undoubtedly be contentions: contention to memory bus, contention for memory ports, contention for interrupt dispatcher, contention for access to kernel tables pro-

tected by mutual exclusion. It must be shown how this contention is avoided or reduced by an appropriate design and how it can be limited with predictable timings.

No low priority service should increase the latency of a higher priority service by holding, acquiring or preempting a processor or any resource in an non essential case nor by causing unnecessary context switches.

Deadlocks caused by dynamic resource allocation must be prevented in the kernel.

1.2.3 Software Engineering Tools

Large real-time applications require a huge engineering effort, both on software and hardware. One might say that real-time is an iceberg with an immerged part of engineering techniques and with a visible part of challenge for guaranteeing timing constraints.

Real-time applications are often very large, require an almost zero default software (some part of it must sometimes be certified), have a long lifetime during which the users requirements change and cause the software and hardware to evolve. For good engineering, all these considerations require facilities for integrating the standards and for using high technology in software engineering.

a-Cross engineering

One usual response to this situation is to use a separate powerful set of design and programming tools and a cross compiler for generating code for the target machine. This fails for real-time applications since:

- there is no standard real-time operating system and the cross compiler needs to be associated with ad hoc libraries for the target operating system,
- program debugging must be started with a real-time and environmental simulator and can be finalized only on the field,
- program maintenance and evolution require the same effort and the simulator has to be adapted when the environment evolves.

b-Integrated engineering

A better approach, using modern computing facility expansion, is to consider the whole engineering of the real-time computing system as

a unique process including creating, debugging, testing, maintaining and evolving the code as well as running the application. Thus the unique artefact which implements this process must be able to support its own workbench for accepting relevant tools.

For this integrated engineering requirement, a real-time operating system should comprise extended capabilities for software engineering. The to-day solution is to propose full Unix compatibility (and IEEE POSIX 1003.4 standard interface [Posix 1992]).

1.3 Basic Concepts of Dune_iX

The basic concepts of Dune_iX cope with the real-time problems presented above and apply to periodic and aperiodic tasks with deadlines to fulfill.

According to this approach, three basic decisions led to the Dune_iX real-time environment:

1) a full real-time behaviour provided by:
   a) a high level real-time task scheduler controlling the selection of tasks and the choice of their priority value, according to the required execution times and deadlines,
   b) a short term scheduling policy provided by a priority driven real-time kernel. This reentrant, preemptive and modular kernel allows several interrupt routines and several tasks to share resources, to switch and to communicate. Immediate task association both with an interrupt level and with an application task context, inter-task shared memory segments, priority inheritance, deadlock prevention, contiguous files are some other relevant features which are provided to reduce all known kinds of latencies due to processors, resources or I/Os contentions.
2) multiprocessing capabilities fully and symetrically implemented in the kernel, allowing to efficiently use a common memory multiprocessor tightly coupled with real-time I/O controllers.
3) full Unix compatibility, i.e. binary and code compatibility.

Beside the specific aspects of Dune_iX, the basic frame-work of its design rely on classical techniques, such as preemption, reentrance,

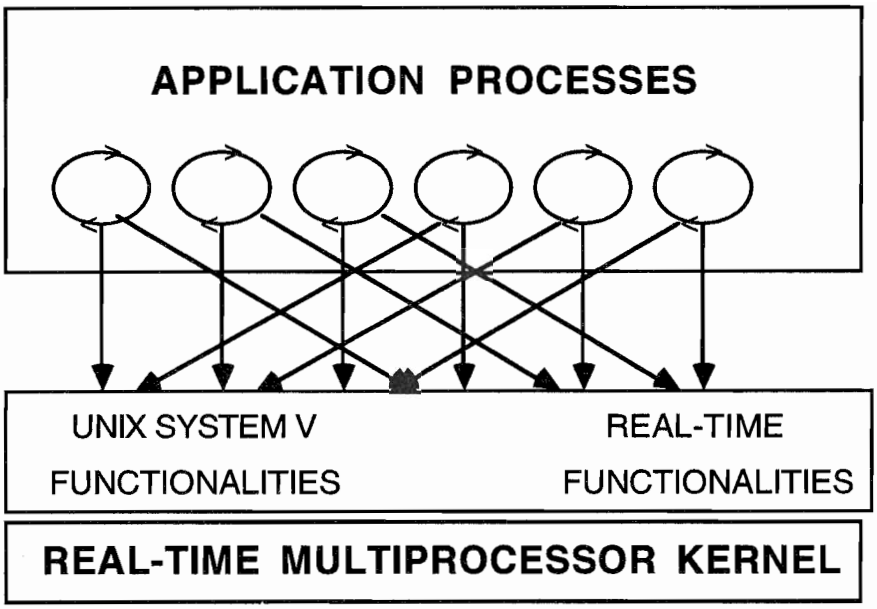


Figure 1. The Dune_iX kernel and its interface

symmetrical multiprocessing, mutual exclusion, deadline scheduling, priority queues, that are being implemented since at least two decades in operating systems [Bétourné 1970, Organick 1972, Wilkes 1970].

These design decisions provide an actual real-time kernel implementing all the Unix functionalities. (Figure 1)

The Dune_iX kernel was first developed in 1982 by a research group of CEA, the French Atomic Energy Research Institution, which decided to implement a new real-time kernel with full Unix interface. It was a Motorola M 68000 uniprocessor version. The current version is the multiprocessor symmetrical version of it which has been engineered by Dune Technologies on Motorola MC 68030s (Dune 3000 architecture).

Related work are found in other recent systems such as LynxOs of Lynx Real-Time Systems, RTU of Concurrent Computer, HPUX of Hewlett Packard, Real/IX of Modcomp [Bauer 1990, Gallmeister 1991, Posix 1991].

2 Real-Time Aspects of the Dune_iX Kernel

2.1 Tasks

Application programs are described in terms of tasks which are logical units of concurrent processing. With each task is associated a priority. The priority value is defined by the user program and this value is used for controlling task scheduling.

Tasks may be **immediate** or **autonomous.**

Immediate tasks are used for reflex reactions to interrupts; their execution is one-shot and non-cyclic, and will not use blocking kernel calls. Once an immediate task has been triggered on a processor, it executes on this processor until completion.

Autonomous tasks have no such restrictions.

An autonomous task can dynamically change its priority value or the priority value of another autonomous task; the kernel cannot do it, unless for coping with priority inheritance.

A priority is given to a task when created. However, the priority level of any task can be modified by the primitive **rt_nice(pid,p)** which enables the running task to set priority p to any task known by its pid.

This allows a particular autonomous task to implement a long term scheduler, which is specific to a given application for coping with special constraints such as periodic hard real-time tasks and which dynamically controls the priority values of these periodic tasks.

The priority value applies to any queueing service in the kernel: scheduling queues for application code as well as for kernel code, semaphore queues, I/O queues or any event queue.

All tasks are preemptive in any execution mode, in application code as well as in kernel code.

2.2 Priority Management

Tasks are ordered according to a fixed hierarchy of priority levels. (Figure 2)

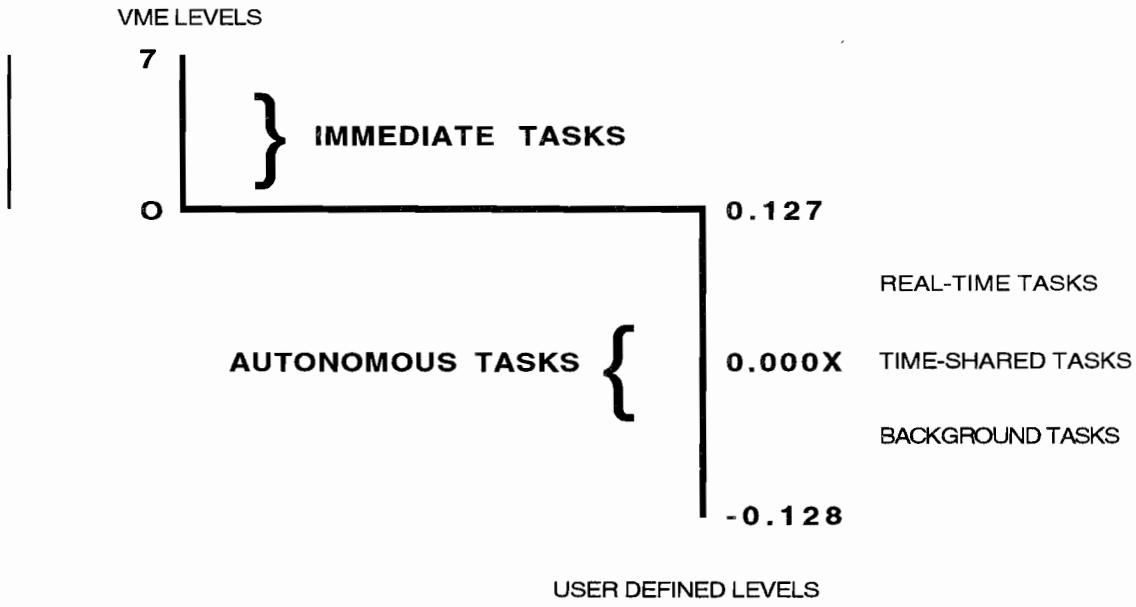


Figure 2. Priority management in Dune_iX

Immediate tasks can receive only one of the seven highest levels labeled 7 to 1, and corresponding to hardware interrupt levels.

All autonomous tasks run with interrupt level 0 and are totaly ordered in a decreasing priority range from +0.127 to +0.000 and from −0.001 to −0.128. Positive values are used for real-time tasks while negative values are used for background tasks. Priority value 0.000 (also written 0.0) is used for the whole set of time-sharing tasks which may receive also a sub-priority level x according to a time-sharing policy; thus the priority 0.000x may be used for UNIX tasks which subpriority level only, x, can be modified for the time-sharing monitoring.

A task is in one of two logical states: **active** or **blocked.** A task is blocked either by a semaphore when a shared resource is not available or when it waits for some result or some signal from another task, or when it has not been activated yet by an external or timing event. An immediate task is never blocked and is created in the active state when its associated event is caught by the interrupt mechanism.

As the number of active tasks may be larger than the number of processors, all active tasks may not run simultaneously; thus an active task may be in one of two substates: **eligible** or **running.**

Immediate tasks are scheduled by hardware and never migrate.

The basic scheduling rule for immediate tasks is the following: when an immediate task becomes eligible, i.e. when it is triggered by an interrupt, it preempts any task of lower hardware priority running on any processor.

The basic scheduling rule for autonomous tasks is that at any time, in a Dune multiprocessor architecture with n processors not preempted by immediate tasks, the n active autonomous tasks of higher priority run.

The scheduler algorithm operates on a single list of active autonomous tasks. This list is organized by decreasing priority. The n running tasks are the first ones of the active list. (Figure 3)

To fulfill this basic rule, it is necessary that autonomous tasks can be preempted and be able to migrate.

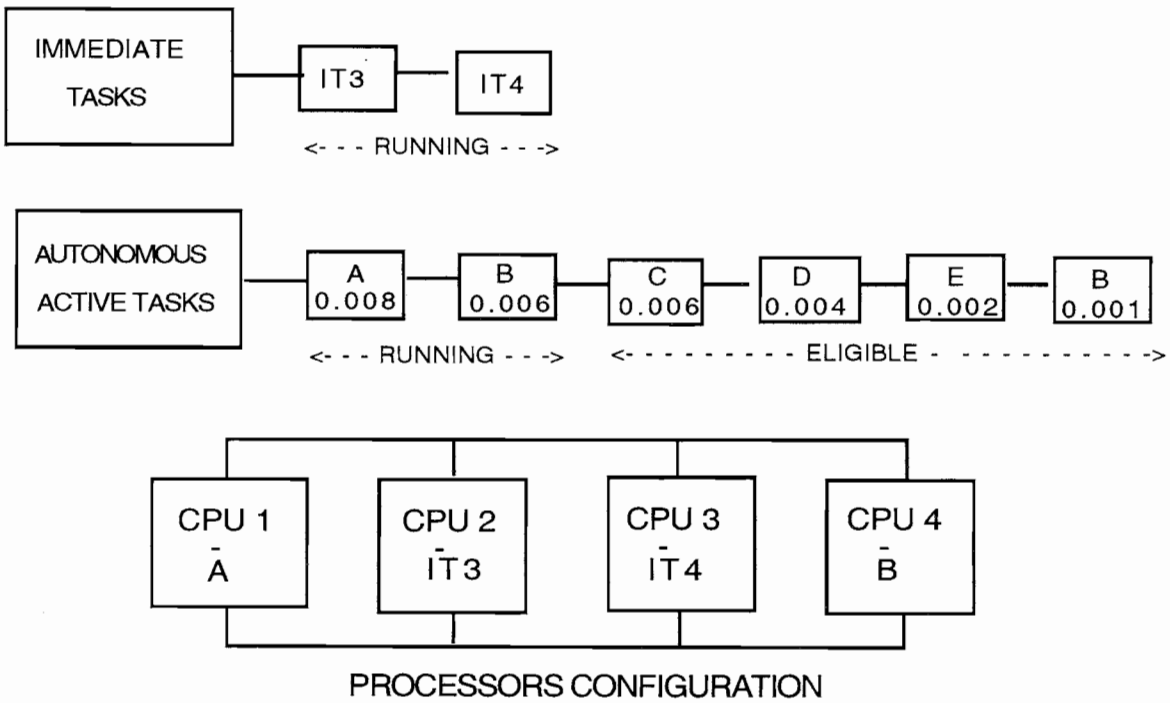


Figure 3. The Dune_iX scheduler active list

Any change of priority or any modification in the set of active autonomous tasks may cause to preempt running tasks.

Some events may modify the list of active autonomous tasks and immediately modify the tasks allocation. These events are:

- blocking and reactivating an autonomous task attempting to access a shared kernel resource (peripheral, system buffer, message, pipe, semaphore, . . .),
- blocking an autonomous task waiting for an interrupt (end of system I/O, signal notification by an immediate task),
- changing explicitly the priority of an autonomous task with the **nice** or **rt_nice** command,
- end of the time sharing time slice or start or end of a delay primitive.

These events lead the running autonomous task t to call the reentrant scheduling kernel routine. At the end of its execution by processor x, this call can have several results, according to the basic scheduling rule and to changes in the set of the nth highest priority tasks:

- return to the calling task t if it is still one of the nth higher priority tasks,
- if the calling autonomous task t is no longer in the set of the nth higher priority active tasks, put the calling task in the list of eligible tasks, allocate processor x to the highest priority eligible task u, force a local context switch of processor x from t to u, and return to the newly current task u,
- if the set of then nth highest active tasks has been changed and nevertheless contains the calling task t, interrupt processor y which is running the active task w of lowest priority and return to the calling task t. On processor y, this interrupt will cause the active task w to call the reentrant scheduling kernel routine and will end with a processor switch of processor y after putting task w in the list of eligible tasks and electing the highest priority eligible task u.

Thus the reentrant scheduling kernel routine is called:

- either explicitly in the code of a kernel primitive, after having inserted, moved or deleted one or several tasks in the list of eligible tasks,

- or indirectly when requesting a shared kernel resource,
- or when returning from an immediate task which has inserted another task in the list of eligible tasks. In the latter case, the reentrant scheduling kernel routine execution is forced at the bottom of the interrupt stack; this causes the immediate task to end with a switch to a task chosen according to the basic scheduling rule and not necessarily to the previously interrupted task.

Some examples of tasks allocations to the different processors are given in Figure 4 [on p. 442].

As immediate tasks do not migrate, they may be forced to inherit the order of their arrival. An illustration of such a situation is given in Figure 5 [on p. 444].

2.3 Task Structures

2.3.1 Task Segmented Space

When considered as a data structure or an information management object, a task is a composite object which is split into six components of contiguous references. These components are:

- a static data segment,
- an invariant text or code segment,
- a stack segment,
- dynamic data segments, allocated at run time, by **rt_alloc,**
- shared memory segments, used for intertask communication,
- an external segment, used for external objects accessing.

The first three segments are standard Unix segments. The others are specific to Dune_iX.

The external segment is managed only at run time with the "**rt_vme()**" primitive which enables or disables access to the entire external (or VME) space. Enabled external addresses are usable only in supervisor mode of execution.

2.3.2 Mapping to Physical Memory

The mapping of the task segments to physical memory is hardware dependent.

When the hardware architecture provides a segmented memory, the mapping is straightforward.

However, in most processors, to-day, a paged virtual memory is managed by a memory management unit (MMU) and a page table. Each task can be bound to a page table and the running task makes use of the MMU. The virtual memory is a single and contiguous address space in which all the different segments of a task must be placed either by the compiler or by the linker. The different segments may be associated with different protection keys. The placement and the protection of segments are done with a granularity of one page. Finally, at run time, the MMU mechanism maps the virtual pages of the running task to the physical page frames which have been allocated to it. In order to share data among tasks, all the virtual pages which are supposed to reference the same data have to be mapped to the same physical frame.

In order to favor response time, the programs are resident and the virtual pages are never swapped; paging is used for placement only.

Still in order to favor response time of disk transfers, logically contiguous pages, corresponding to a memory zone allocated by **rt_alloc** or to a memory segment, are mapped into contiguous physical page frames. Thus an I/O transfer need not use the MMU nor be aware of page boundaries.

Memory management of the Dune 3000

In Dune 3000, a host architecture presented in chapter 3 below, every task is bound to a virtual memory which is divided into pages of 4 KBytes. The virtual memory is partitioned in a main subset which is mapped to physical memory by a MMU mechanism and in a VME subset which is mapped to the real time I/O boards of the Dune 3000 computer.

The MMU mechanism of each processor board translates the virtual address into a physical one. The physical memory of the Dune 3000 can vary from 8 to 48 MBytes, depending on the configuration, and is divided into page frames of 4 KBytes.

2.3.3 Autonomous Task Creation

Creation of an autonomous task is the result of the activation of the "**fork()**" primitive, leading, as in UNIX, to the duplication of the static data segment, the stack segment and the dynamic data segment. The text segment and the shared memory segment are shared.

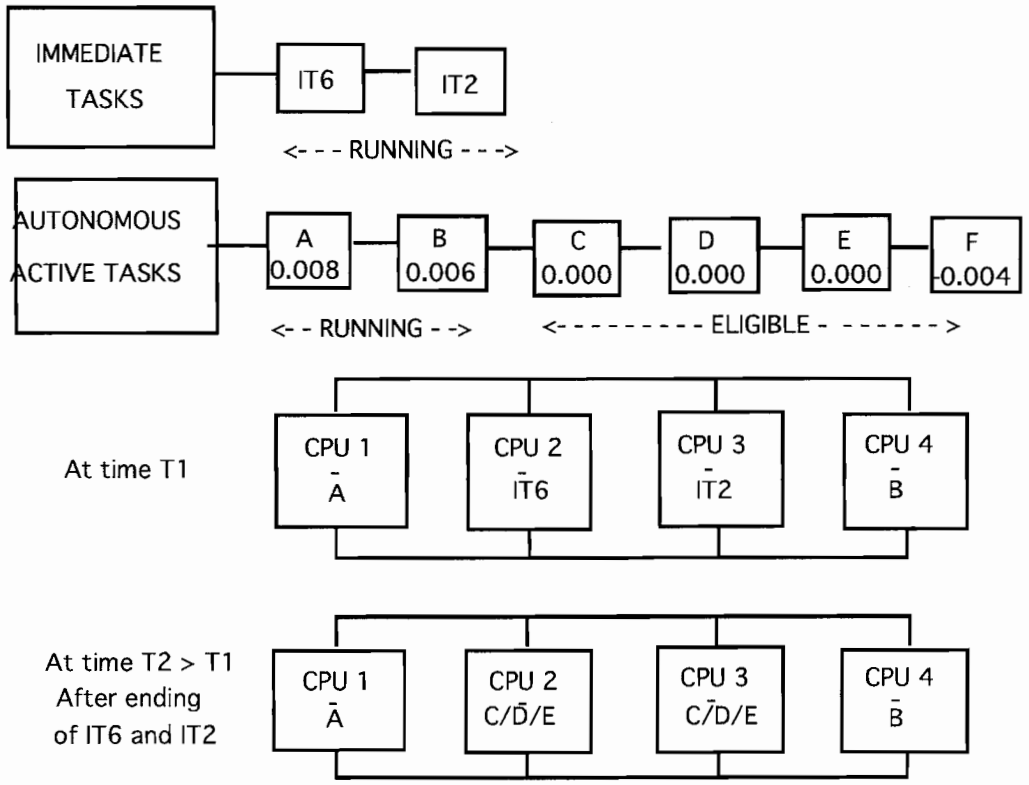


Figure 4. Examples of task allocations in Dune_iX a) First example of task allocation

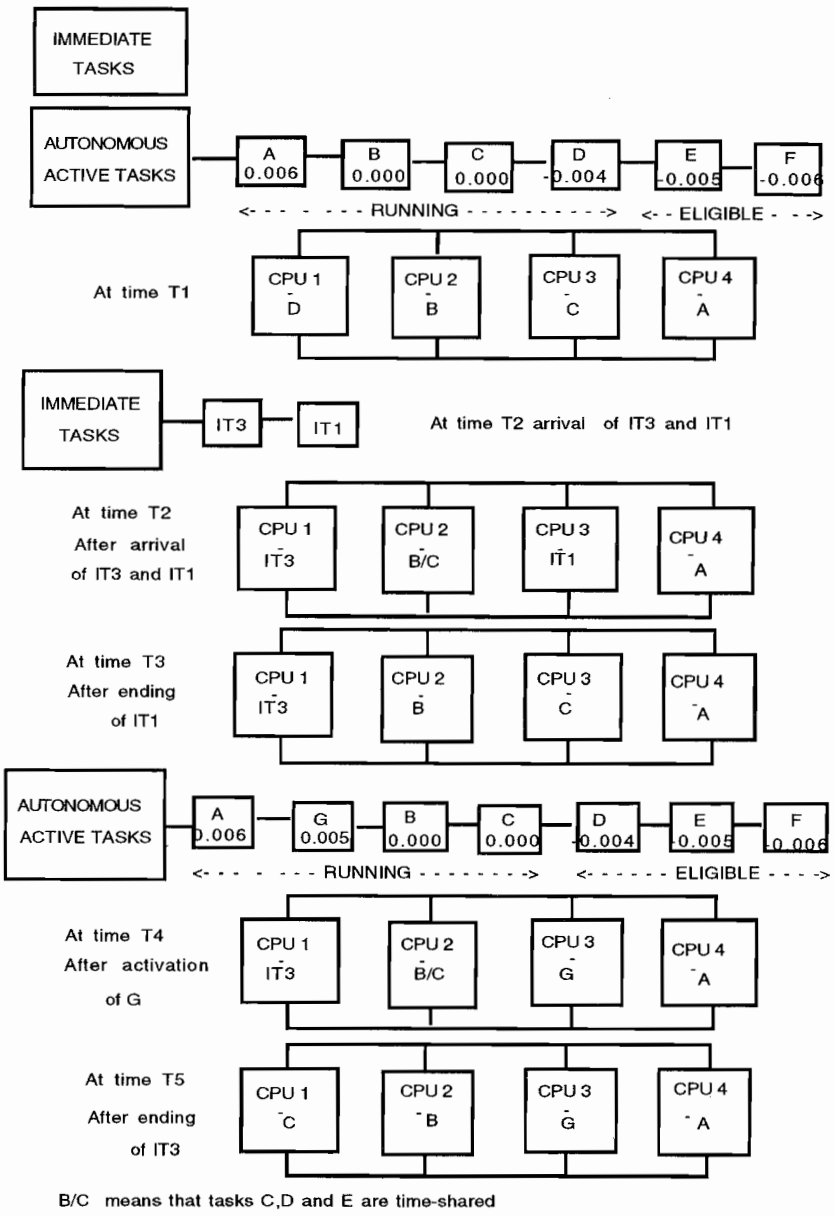


Figure 4. b) Second example of task allocation

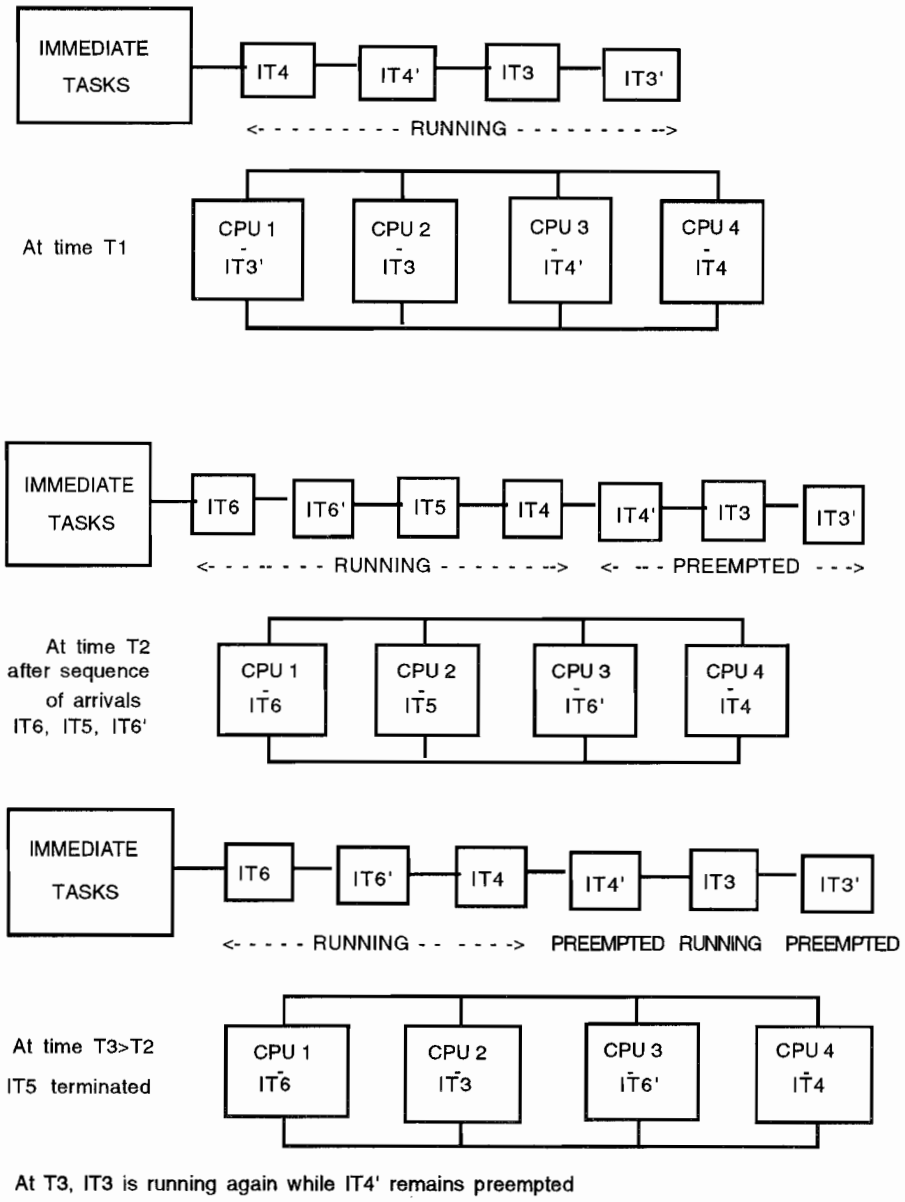


Figure 5. Consequence of non-migration of immediate tasks

2.3.4 *Multithreading and rt_fork*

A primitive for creating real time multiple threads of control is also available. The "**rt_fork()**" primitive creates an autonomous task with the same effect as "fork", except that it duplicates only the stack segment. The parent and child resulting tasks therefore share their code, static data and shared memory segment. This latter segment can then be used also for shared memory data communication schemes. (Figure 6)

2.3.5 *Immediate Tasks Declaration*

When an immediate task is declared, it is associated with an execution context consisting of an external interrupt vector value, of a private interrupt stack and of a procedure which should be executed when the interrupt occurs. An immediate task is declared within an host autonomous task the virtual memory of which acts as a shared carrier. The procedure code of the immediate task is placed in this virtual memory. When the immediate task is created on a given processor, it runs bound to its private interrupt stack and to its carrier virtual memory. Several immediate tasks may be declared within the same carrier virtual memory. Moreover, in a Dune multiprocessor architecture with n processors, all n processors must be available to respond to n occurrences of the same associated external interrupt; thus n immediate tasks must be created. This is made feasible by providing reentrant interrupt procedures and by associating n execution contexts with an immediate task declaration, each context providing a private interrupt stack for each processor. (Figure 7)

The following kernel primitives are available for immediate tasks (Figure 8):

rt_imt(routine_address, int_vector, max_stack_size) allows an autonomous task to create as many immediate task contexts as processors; each context is associated with the external interrupt vector int_vector value, the autonomous task pid, a stack of size max_stack_size and the interrupt procedure routine_address. The priority level is fixed by a bus controller register. An immediate task runs in supervisor mode and consequently without any system protection.

rt_delimt(int_vector) allows to suppress the immediate task contexts associated with the external interrupt vector int_vector.

rt_fastimt(routine_address, int_vector) allows to create express immediate tasks which will run without context saving; if the interrupt

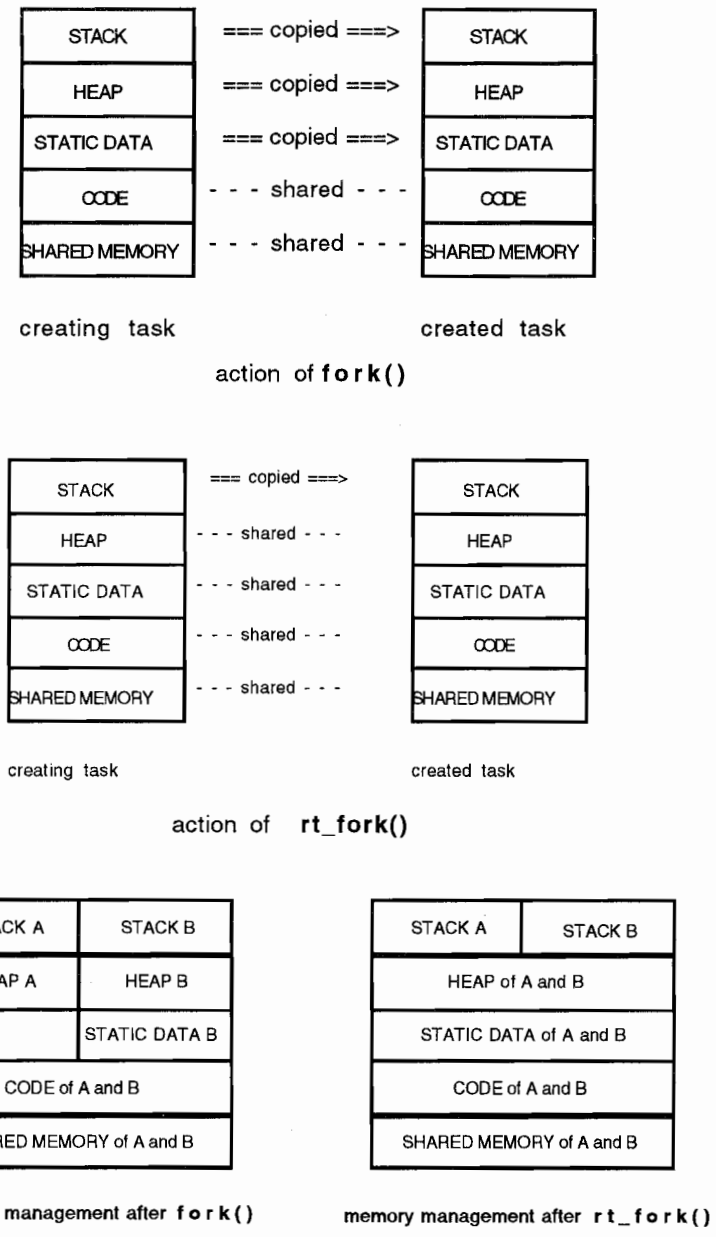


Figure 6. Autonomous task creations in Dune_iX

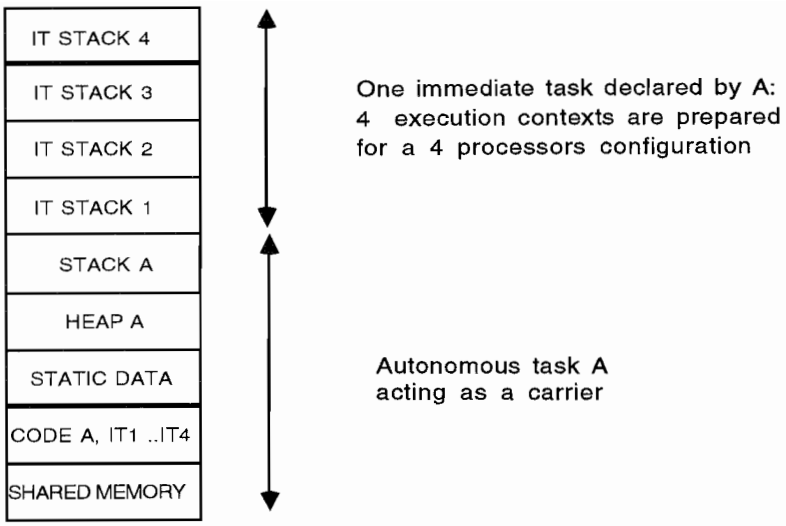
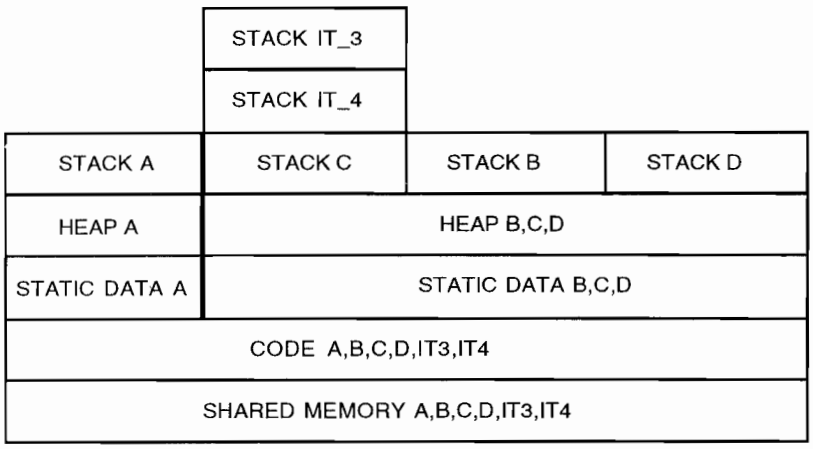


Figure 7. Memory management for an immediate task




```
main( )
.
if (fork( )==0)   proc_A( );
if (rt_fork( )==0) proc_B( );
if (rt_fork( )==0) proc_D( );
rt_imt(proc_IT_3,  );
rt_imt(proc_IT_4,  );

proc_C.
```

Figure 8. A example of programme and its memory management on a uniprocessor configuration

procedure routine_address does use registers it has to be programmed to explicitly save them; thus an express immediate task has a faster context switch.

Note that the sole primitives which an immediate task may use are the non-blocking following ones: **rt_notit(), rt_valid(), time(), rt_time(), kill(), signal(), msgsnd(), semop(),** and **exit().**

2.4 Task Cooperation

2.4.1 Interrupt and Immediate Task Synchronization

Any external interrupt causes the creation of an associated immediate task. This immediate task can read or write the external bus devices. It can also trigger another external bus interrupt and cause the creation of another immediate task which may run on another processor or on the same processor after preempting the former immediate task, according to the basic scheduling rule for immediate tasks.

Note that an immediate task may trigger an interrupt at the same level, creating a clone task which may run concurrently on another processor, if it is available, and provide additional I/O transfer power. (For example, if an immediate task has to read a large vector of external devices, one immediate task may read it upwards after creating a clone to read it downwards, each task ending when all the devices have been read, which is noticeable with a version number or a reading date.)

2.4.2 Immediate Tasks Cooperation Inside the Kernel

Immediate tasks may call reentrant kernel primitives and may use some shared kernel resources; their cooperation as usual in any symmetrical multiprocessor architecture, uses a kernel spin lock, which sets processors in a busy waiting loop when necessary. This spin lock is controlled by two procedures **mit()** and **rit()** which are executed with interrupts masked. (see Illustration 2)

2.4.3 Immediate Task and Autonomous Task

The interactions between immediate tasks and autonomous tasks are the following ones (Figure 9):

a-an immediate task has access to the context of its host autonomous task and it may read or write data in the shared communication segment of its virtual memory.

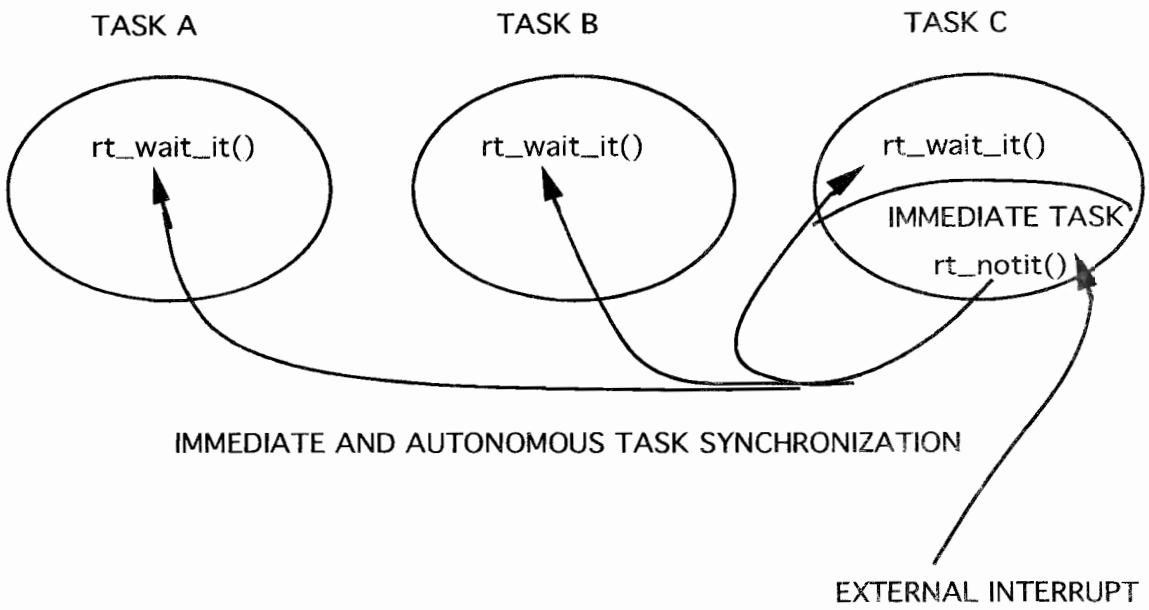


Figure 9. Synchronization of immediate and autonomous tasks

b-a direct synchronisation mechanism is available between the two types of tasks with a kind of kernel counting semaphore, also implemented with spin locks. For this purpose, an integer count is created for each immediate task. The kernel primitives are the following:

rt_notit() is used by an immediate task to post a signal; it acts like a V primitive with semaphores; the associated count is incremented and, if any, the highest priority queued autonomous task is activated;

rt_waitit(int_vector, delay_value) is used by an autonomous task to wait for a signal posted by an immediate task; it acts as a P primitive with semaphores; the count associated with the immediate task coping with the interrupt int_vector is decremented and if it is negative the calling autonomous task is blocked and priority queued; the task is blocked at most a number of milliseconds fixed by the parameter delay_value.

rt_valit(int_vector) indicates the current value of the count associated with the immediate task coping with the interrupt int_vector.

c-this direct synchronization mechanisms extend straightforwardly. On the one hand, the rt_notit signal may be provided by any task in the set of immediate tasks sharing the same interrupt vector value int_vector. On the other hand, any task in the set of autonomous tasks sharing a given uid (user identifier) may receive this signal, or wait for it, when using rt_waitit.

d-an autonomous task can trigger an immediate task by simulating a external interrupt with **rt_simit(int_vector, int_level).**

e-standard Unix System V ipc primitives with messages, signals or semaphores are also available with the restriction that immediate tasks can call only **msgsnd()** and **kill()** primitives or carry out only V operations on semaphore sets.

2.4.4 Autonomous Tasks Cooperation

All data communication, cooperation and synchronization among autonomous tasks are performed with kernel primitives which present the standard Unix semantic and interface and which have a real-time reentrant and preemptive implementation.

These are the standard unix ipc (messages, semaphores, shared memory segments), pipe, signal or network (sockets, streams) primitives (Figure 10).

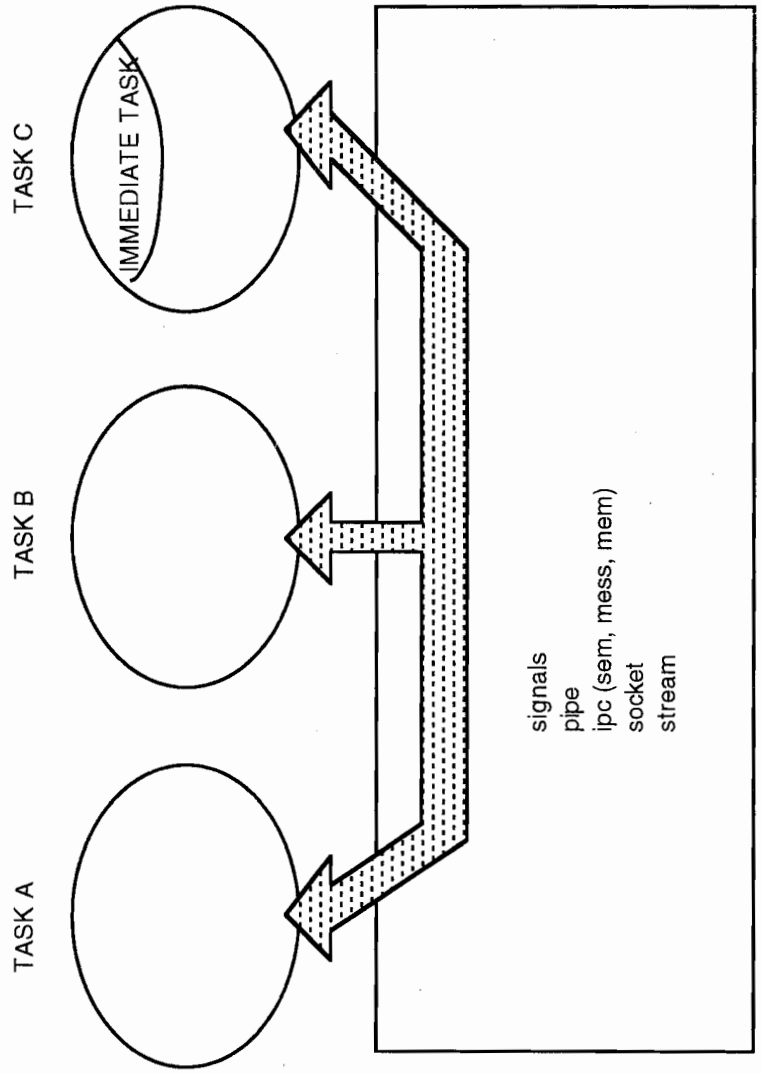


Figure 10. Tasks communication

2.4.5 Synchronization Within the Kernel

Since the kernel is reentrant, several kernel calls, resulting either from interrupting a processor or from multiprocessing, may compete for the shared but mutually exclusive resources. This sharing is done in a classic way by kernel semaphores using the spin locks presented above (see Illustration 3). Recall that any task queueing and queue service is priority driven and that all servers are preemptive, except during the spin lock service (compulsory either for data consistency and for deadlock prevention).

2.5 Kernel Resource Management

2.5.1 Critical Sections

A kernel implements several objects such as tasks, virtual memories, pipes, events, files, and so on, and the descriptors of these objects can be considered as logical resources; similarly the descriptors of peripherical devices and of pages of physical memory are considered as resources. The Dune_iX real time kernel implementation is not monolithic. Careful design can lead to clean separation of the resources allowing concurrent calls of the kernel as long as these concurrent calls do not use the same resource. In a concurrent kernel, each resource or each class of resources has to be kept consistent and its access has to be protected by a critical section. Critical sections of code are programmed with spin locks (if necessary, a kernel call is forced in a busy waiting loop) or with kernel semaphores (if necessary a kernel call is blocked and the processor is allocated to another task or to another kernel call).

2.5.2 Deadlock Prevention

Suppose that two tasks T1 and T2 need both two resources R1 and R2, which are each mutually exclusive. For the sake of consistency suppose that T1 and T2 are not preemptive. This may cause a deadlock if T1 requests successively R1 and R2, and if T2 requests successively R2 and R1. It occurs when T1 is waiting for R2 already allocated to T2, and when T2 is waiting for R1 already allocated to T1. This deadlock situation can be generalised to any circular wait.

In the example above, deadlock will never occur if both tasks request successively R1 and R2. This scheme can be generalised in programming all resource requests in the same order; this is the classical

policy of deadlock prevention by ordering all the resources (it was originally devised by Havender in 1968 for IBM OS/360 [Havender 1968, Habermann 1976]).

In the case of a small size kernel with an a priori known list of resources, this programming policy is easily manageable; this has been done for the Dune_iX kernel. This policy is also usable and should be recommended for any real-time applications with fixed set of resources.

Recall that the same deadlock situation occurs when an immediate task running a critical section of code on a processor P is interrupted for allocating the processor to another immediate task which in turn needs to run the critical section. This situation is avoided by masking interrupts during the critical section of code (see the spin lock programming above).

2.6 Reducing Kernel Latency or What Preemption Really Means

2.6.1 Kernel Reentrance Plus Kernel Preemption

The Dune_iX kernel is not monolithic and therefore its design allows several kernel calls to concurrently run while using different resources. This allows a multiprocessor system to have a unique kernel code in its common memory and to run this kernel code purely symmetrically. To be effectively concurrent, the kernel code need to be reentrant and to use a stack per kernel invocation; the stack used is that of the invoking task which is used both for user code and for kernel code.

Once reentrant, the kernel can also be preempted without any additional cost since spin locks and semaphores have already been implemented for reentrancy and multiprocessing. Moreover, a kernel call can start running on a processor, be preempted, and finish running on another processor. This allows strong symmetry of processor utilization [Habermann 1976, Organick 1972, Wilkes 1970].

2.6.2 Priority for Kernel Invocations

Once the kernel being reentrant and preemptive, there is no difference between kernel code and user code; therefore the kernel code, which is running a kernel call invoked by a task T, is just a particular procedure of that task T, and is part of the task (it needs not to be run in supervisor mode unless it executes privileged instructions or accesses

protected data); it uses the task stack and it shares all the tasks behaviour. Thus it can be ruled by the task priority mechanism. Blocking a kernel call with a semaphore is just blocking the calling task; reactivating a blocked kernel call is just reactivating its calling task. As a consequence, a low priority task running a kernel call will be preempted by a high priority task running user code. The scope of the priority policy contains the kernel.

2.6.3 Priority Inheritance

Critical sections and priority scheduling can lead to a paradoxal phenomenon [Kaiser 1983, Sha 1987] known as priority inversion.

Consider, as an example, a monoprocessor system with four real-time tasks, T1, T2, T3 and T4 with decreasing priorities. At a given moment they are all blocked except T4 which is running a critical section. Suppose now that T1, T2 and T3 are activated by an external event. The basic scheduling rule causes T1 to preempt the processor. Suppose that T1 needs to run the same critical section as T4. The mutual exclusion semaphore will block T1, and the basic scheduling rule will elect T2, and after it, T3, for running. They run before T1, that has a higher priority. When they end, then T4 can run and finish the critical section. Only then T1, the most urgent task, can proceed!

One method of limiting this effect is to use priority inheritance [Kaiser 1983]. With priority inheritance, a task priority is no longer static; if a task T1 is blocked waiting for task T4 to undertake some computation in a critical section, then the priority of T4 becomes the maximum of T1 and T4 priorities, as long as T4 remains in critical section. In other words, a task within a critical section dynamically inherits the maximum of the priorities of the tasks queueing for this critical section if this maximum is higher than its present priority. When T4 quits the critical section, it receives back the priority it had when entering the critical section.

Note that priority inheritance is dynamic: when a new task is queued, it may contribute to increase anew the priority of the task running the critical section.

Note also that critical sections are often nested; therefore priority inheritance has to be recursively applied: when a new task is queued, it may contribute to augment the priority of the task running the critical section and, if this latter task is queued for a second critical section,

the priority of the task running this second critical section may also be augmented. When a task is queued temporarily only or when a signal causes its abortion, the inheritance has to be cancelled recursively.

Priority inheritance is of paramount importance for real-time systems and is implemented in Dune_iX. Priority inheritance applies recursively in Dune_iX.

2.7 Real-Time Input-Output

2.7.1 Real-Time Disk and Contiguous Files

The standard disks are partitioned, and each partition can receive a Unix type "file system", which includes all the types of "Unix files", (normal, directory, special, pipe, link) plus the "contiguous file" type. The disk cache is handled by the I/O controller.

Dune_iX kernel enables to declare contiguous files. A contiguous file is a file in which logically consecutive blocks have been mapped onto physically consecutive blocks on disk. Thus a single disk access can transfer several contiguous blocks of a file to a user memory buffer. The benefit of such files is that they require less head movement and have shorter access time. Applications have a better control of the access time to contiguous files. Those files are created by a particular primitive **rt_create(),** and are then used with any standard Unix primitive such as **open(), read(), write(), stat(), exec(). . . ,** and of course by utilities using these primitives.

Contiguous files can be used to contain the binary executable images in order to speed up loading. Of course, Unix utilities are good candidates for contiguous files.

Each opened file contains one single buffer in the kernel, which includes the current block being used by the application.

2.7.2 Direct Disk Input-Output

Input and output operations with contiguous files may be directly performed with a task buffer, without passing via a kernel buffer when using the particular **rt_dread(), rt_dwrite()** primitives. So, when the current pointer is positioned on a disk sector border, and when the request is an integer number of sectors, there is only one request transmitted to the I/O controller, which moreover performs seeking and accessing in one operation.

2.7.3 Disk Consistency

When a task requests successive logical write operations on a disk or when a logical write request is transformed into several physical write operations on the disk, the latter are always performed in the same order as the logical requests. There is no optimization on transfers on a given disk. This is done to ensure file system consistency. Therefore, in case of power supply breakdown, the "file system description" remains consistent with the stored status of the disk; this allows to recover a consistent disk space after failure. The absence of write optimization at the disk transfer level may slow down the standard Unix file operations. If these operations are time critical, then contiguous files should be used.

2.7.4 Real-Time Behaviour of Device Drivers

Dune_iX allows to incorporate new device drivers within the kernel. Any device, for example a disk driver, is a kernel object which is accessed by a set of procedures and which is described by a device descriptor. Device procedure are reentrant code. As any kernel object, the device descriptor, and the device procedures, may be shared by several tasks and its access is therefore protected by a specific semaphore. As the semaphore queue is ordered by priority, the I/O operations are also priority driven.

The I/O operation use the maximum available parallelism since each device is a separate resource; if several disks are physically accessible in parallel, the kernel may contain as many resources, i.e. as many disk descriptors, as there are distinct paths to disks.

2.8 Time Management

The standard Unix timers have a time granularity of one millisecond and are used in **alarm()** and **rt_time()** primitives. Additional timers are present and are programmable by applications. They are based on five 16 bits physical timers which can count with binary or decimal digits, upwards or downwards. Two of them can generate interrupts.

2.9 Integrated Engineering

Dune_ix is a kernel which provides a fully Unix compatible interface. All system calls of Unix System V Release 3 are basic Dune_iX calls

and they are executed by the Dune_iX Real-time kernel with real-time behaviour. Other kernel calls have been added for specific real-time functionalities.

The whole set of kernel calls is illustrated in Figure 11.

For the Dune 3000, the Unix compatibility has been extended to binary compatibility with the Motorola Unix system, SYSV68K. This allows to directly use all binary versions of utilities developed for the latter system, such as networking facilities, X-window graphics or de-

UNIX SYSTEM V FUNCTIONALITIES

accept*	fork	msgget	semget	stime	uname
access	getdents	msgop	semop	sync	unlink
alarm	getmsg	nice	send*	sysfs	ustat
bind*	getpid	open	setpgrp	time	utime
brk	getsockname*	pause	setuid	times	wait
chdir	getsockopt*	pipe	shmctl	uadmin	write
chmod	getuid	plock	shmget	ulimit	rmdir stat
chown	ioctl	poll	shmop	umask	select*statf
chroot	kill	ptrace	shutdown*	umount	semctl statfs
	close	link	putmsg	signal	
	connect*	listen*	read	sigset	
	creat	lseek	recv*	socket*	
		dup	mkdir		
		exec	mknod		
		exit	mount		
		fcntl	msgctl		

*Primitives from UNIX SD

DUNE_iX REAL-TIME EXTENSIONS

rt_alloc	rt_fastimt	rt_notit	rt_valid	rt_vme
rt_creat	rt_delimt	rt_waitit	rt_stat	rt_read
rt_fork	rt_mod	rt_simit	rt_time	rt_write
	rt_imt	rt_ nice		

Figure 11. List of Dune_iX kernel primitives

velopment tools. This also allows to embed new and powerful software tools in real-time applications.

3 Example of a Host Architecture for Dune_iX

3.1 Logical Architecture

This section describes the requirements for an architecture which can enhance the real-time behaviour of Dune_iX.

Such an architecture must rely on two strongly coupled basic units (Figure 12):

- a computing block providing symmetric multiprocessing and common memory,
- an input-output block composed of a set of I/O controllers.

The efficiency of the coupling of elements of each block is provided by two interconnection buses and by a dynamic interrupt router.

a) The computing block

The computing block should comprise a set of processors, possibly with their own cache and local memory, a set of shared autonomous memory boards, an I/O processor transfering I/O data and interrupts, a set of timers and one or several interconnection buses.

The processors should have a test and set like operation, and a MMU device providing memory relocation and protection.

The incoming interrupts should be dynamically forwarded to an available processor.

The computing block should be designed in order to reduce all possible contentions such as contention to memory ports or buses, to the I/O processor, to the interrupt dispatcher, etc.

b) The input-output block

The input-output block should contain a set of I/O controllers providing standard access to real-time devices as well as to working station devices.

These controllers should allow transfers with disks, networks, printers, serial lines, terminals, bit-map graphics.

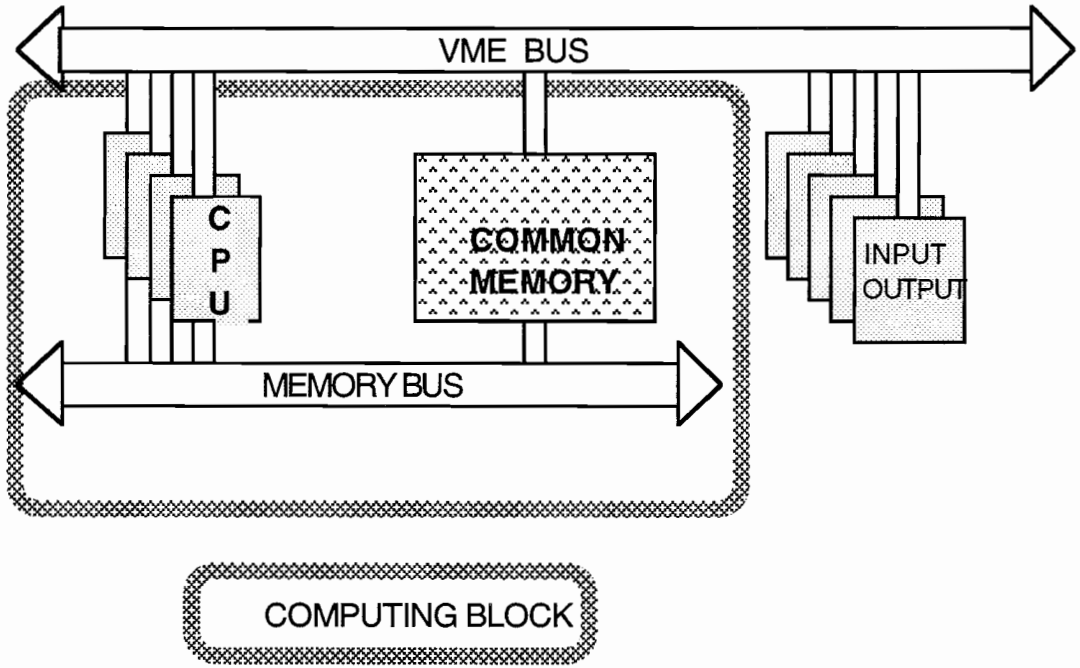


Figure 12. Logical architecture of Dune_iX

3.2 Example of the Dune 3000

The Dune 3000 (Figure 13) is an example of a host architecture which has been designed by Dune Technologies.

The computing block operates on a fast memory bus working at 37,5 Mbytes/s. Processors access memories via this internal memory bus. The memories are shared by all processors and include at any time, Dune_iX operating system as well as applications data and code. This is consistent with the design decision to provide a symmetric multiprocessor architecture.

The input/output boards are connected to a VME bus, (system I/O board, and user's real time I/O boards) and they provide the different connections to the usual peripherals as well as to the real time devices.

a) Processors

The processor boards are equipped with Motorola MC 68030 processors operating at 25 MHz, coupled with the floating point coprocessor MC 68882. The Dune 3000 can operate with 1 to 4 processor boards. None of the processors is specialized.

Each processor accesses the common memory through a private cache of 32 Kbytes. Each processor also has a local memory of 32 Kbytes used by the kernel.

The caches are automatically updated by hardware. Cache consistency is ensured by a "bus snooper" based on the "write through" technique, of a "replace on match" type. Therefore each cache includes a mechanism which detects a write operation made by another processor on the memory bus and which invalidates the corresponding input, if any in the cache.

This way of using the cache memory avoids unnecessary loading of the memory bus. Caching and consistency management are performed by hardware, therefore the cache operations are hidden to the programs.

b) Memories

The memory boards are of an autonomous type, allow fast access and provide a double port on the memory bus and on the VME bus. Each memory board has 8 Megabytes. The Dune 3000 can operate with 1 to 6 memory boards allowing up to 48 Megabytes for the common memory.

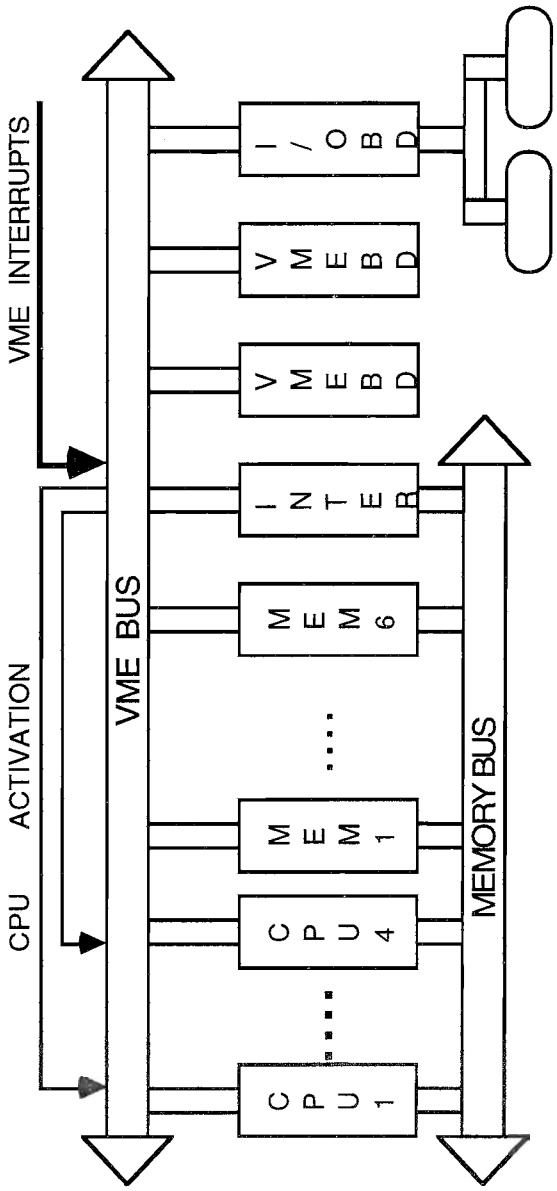


Figure 13. Dune 3000 Multiprocessor

Each memory board is optimized to provide the minimum traffic on the memory bus and to speed up the exchanges between memory and processors. These boards include a simple feature to detect single errors (1 parity error per byte).

The local memory and the cache are static memories without interleaving. Their access is made on 32 bits with zero wait state. Access time is 2 processors cycles (80 nanoseconds).

Single access to the local memory is 320 nanoseconds. In case of cache line loading, the duration is 720 nanoseconds for a quadruple access.

The VME BUS has a 1 gigabyte address space.

c) The "VME" interface board

The interface between the "computing block" and VME peripherals is performed in several ways which all are consistent with the design decision to provide a symmetric multiprocessor architecture.

a) The processors can directly read and write on the VME bus in order to perform input or output operations with the peripherals and the real time controllers.

b) The VME peripherals can directly access any part of the common memory via the VME bus. Write operations are routed to the memory bus in order to activate the cache consistency mechanism. This routing is performed by the VME interface board. Read operations are directly made through the VME port of the memory boards.

c) The VME interrupts are dynamically dispatched by a hardware mechanism located on the "VME interface" board. For each interrupt occurrence, this mechanism selects a processor which has to respond. The selection is based on priorities; thus the chosen processor is the one running the task of lowest priority.

In addition, and again to favour real-time applications, the VME interface board includes two types of timers:

- a FRC type timer (Free Running Counter), of 48 bits, set up with the system and incremented each microsecond,
- two 16 bits programmable timers that rise interrupts when the programmed delays have been achieved.

A processor is able to interrupt another one by writing at a given address in a dedicated register (SI register) on the board. This triggers a sofware interrupt.

Any interrupt risen by a timer or by program has a level and a vector attached to it. These interrupts as well as the hardware interrupts are taken into account by the automatic dispatching mechanism.

The level and the vector of the timers interrupt are fixed when programming the timers. The level and the vector of the software interrupts are determined by the data included in the SI register. One part of this data indicates also whether the interrupt has to be routed to a particular processor, or taken into account by the automatic allocation mechanism.

d) The Dune 3000 input-output block

The Dune 3000 accepts any VME double Europe format (6U) board.

The system input-output transfers with disk, printers or network, are handled by a dedicated controller.

4. Real-Time Capabilities

The appraisal of a real-time operating system relies mainly on real-time capabilities such as:

- promptness of response by the computer system,
- predictability of kernel calls execution times,
- tuning of scheduling policies,
- assistance provided for program debugging in the real-time context when the application is running on the field,
- performance recorded in case studies.

All the following values were measured on the Dune 3000 architecture.

4.1 Promptness of Response

The promptness of the response of a real-time kernel may be evaluated by two numbers, interrupt latency and clerical latency.

a) **Interrupt latency** is the delay which occur between the advent of an event in the application and the instant this event is recorded in

the computer memory. This interrupt latency is caused by:

- the propagation of the interrupt through the hardware components: external bus, interrupt dispatcher, interrupt board of the processor, interrupt selection,
- the latency in the kernel software resulting from non-preemptible resource utilization: masking interrupts, spin lock action,
- the delay for context switching to an immediate task.

In Dune_iX, this interrupt latency is reduced by a systematic use of the hardware priorities of the external bus, by kernel preemptivity and context switch to immediate tasks.

Thus the interrupt latency is of 5 microseconds with **rt_fastimt** and of 15 microseconds with **rt_imt,** with the Dune 3000 implementation of Dune_iX.

b) **Clerical latency** is the delay which occurs between the advent of an event in the application and the instant this event is processed by its target application task. This clerical latency is caused by:

- the interrupt latency,
- the transfer of data from the interrupt subroutine to the application programs context,
- the notification that the target application task is already eligible,
- the return to the current application task, which may be using some non-preemptive resource and, in that situation, must be protected against the election of another application task,
- the delay the target application task waits before being elected for running,
- the installation of the context of the target application task.

In Dune_iX, this clerical latency is reduced by systematic use of software priorities, by priority inheritance, by the sharing of memory between immediate tasks and application tasks, and by two kernel calls, **rt_notit** and **rt_waitit** which allow a producer-consumer relationship between immediate and application tasks. Thus the clerical la-

tency is within the range of 200 and 300 microseconds for the Dune 3000 implementation of Dune_iX.

A detailed analysis of the response time to interrupts is given in Figures 14 and 15.

K_SC_ENTER is a routine of the kernel which is called when entering the kernel by a system call instruction. When the call is **rt_waitit,** the calling context is saved and the interrupt value is noted before calling the task commutation module K_SW.

K_SC_EXIT is a routine of the kernel which is called before leaving the kernel: an asynchronous task context is restored.

The time for executing K_SC_ENTER and K_SC_EXIT had been measured with a void system call and it took 130 microseconds. The part of K_SC_EXIT is 65 microseconds.

K_SR is a routine of the kernel which is forced by an interrupt: it saves the registers of the kernel in the kernel stack, finds out the associated immediate task and prepares its context.

K_RR is a routine of the kernel which is called when leaving the immediate task: it restores the kernel context and returns from the exception.

The time for executing K_SR and K_RR has been measured with a void immediate task and costs less than 25 microseconds. Such a void immediate task has been triggered at a rate of 40000 interrupts per second on a uniprocessor and at a rate of 120000 interrupts per second on a three processor configuration.

K_SW is the kernel routine which performs the task election according to the Dune_iX priority rule. Its execution costs less than 90 microseconds.

K_SC is any kernel routine under execution when an interrupt occurs.

K_IT_MASK is a section of code within K_SC which is embedded between mask_interrupts and unmask_interrupts and which therefore runs with disabled interrupts. Its maximum value is 20 microseconds.

K_PR_DISABLE is a section of code within K_SC which is embedded between invcom and valcom and which therefore runs with disabled task commutation module. For system calls non-concerned with the filing system, its maximum value was 100 microseconds. When files are accessed during the system call, the maximum duration is 500 microseconds.

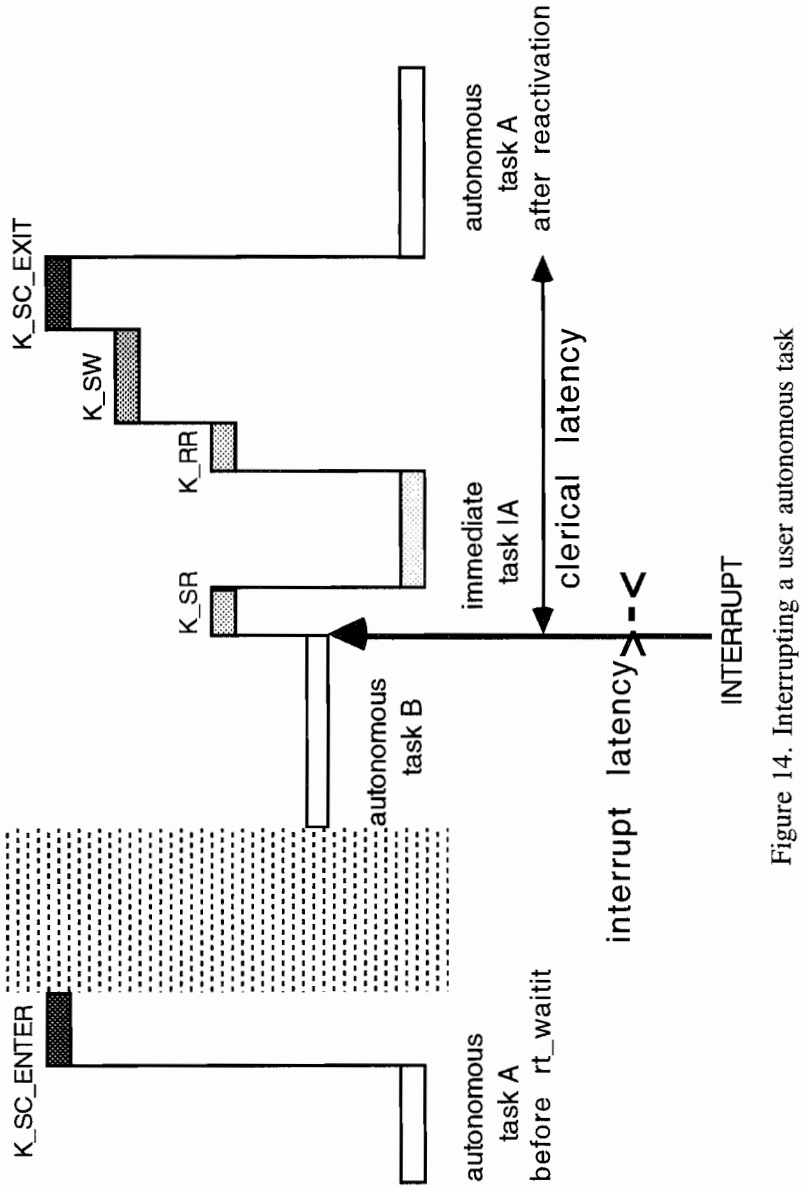


Figure 14. Interrupting a user autonomous task

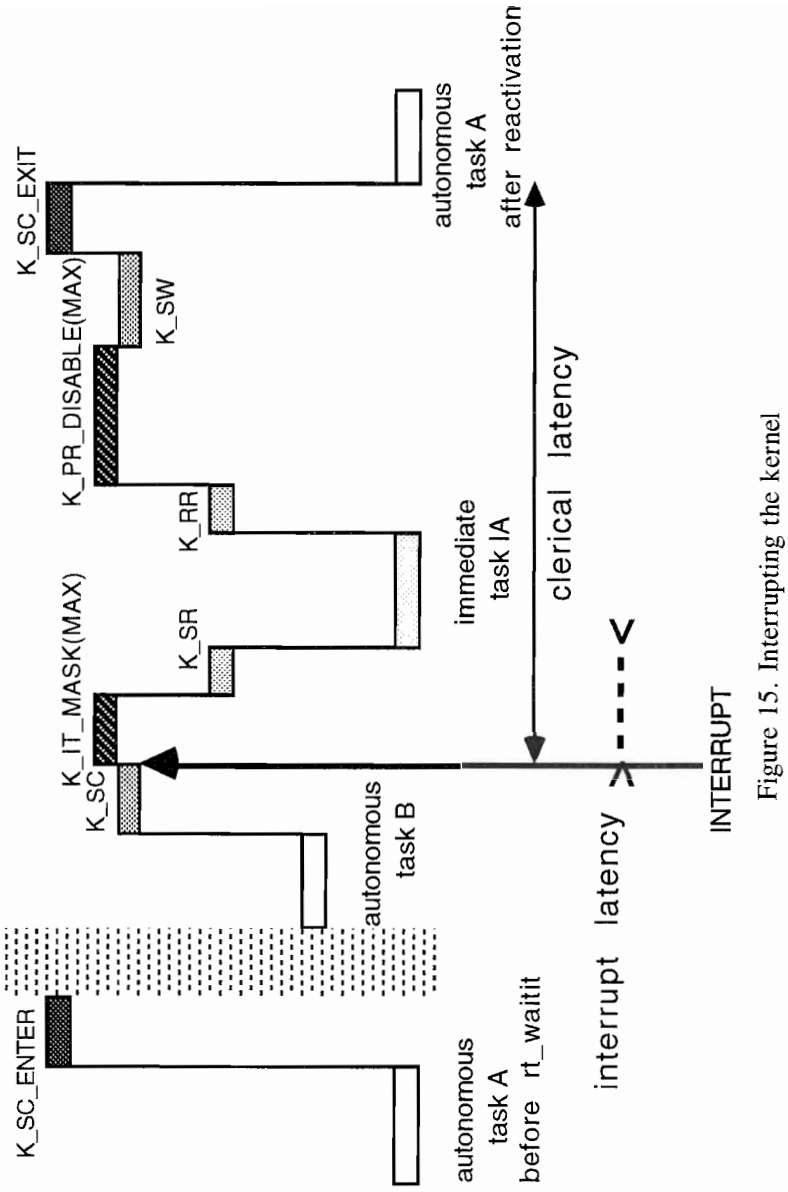


Figure 15. Interrupting the kernel

4.2 Kernel Calls Execution Times

The Dune_iX kernel includes a complete set of methods for reducing time latency which are reentrance, preemption, priority scheduling and priority inheritance.

Therefore the execution time of each kernel calls can be exactly evaluated when it is executed for the highest priority task. This time is that of the call itself plus the delay of the longest critical section in the kernel.

Each kernel call contains:

- K_SC_ENTER and K_SC_EXIT which costs 130 μs on the Dune 3000,
- K_SW which needs 90 μs on the Dune 3000,
- possibly K_PR_DISABLE,
- and the kernel code for its specific action.

A systematic study of all kernel calls is thus feasible for a given host architecture.

4.3 Analysis of Scheduling Policies for Periodic Tasks

A very simple and powerful tool, the periodic simulator, has been implemented for allowing users to simulate on line the load and the real-time behaviour of periodic tasks. Its role is to schedule the execution of several tasks and to verify that they have terminated before a given deadline.

The periodic simulator implements on line a scenario which is periodically and endlessly repeated. The simulator counts the deadline overpasses, and can eventually suggest decisions, such as change of priority of a task which overpasses too often its deadline.

The use of the scheduler is very simple. The user describes the scenario to be executed, in the form of commands and options. This description contains the activation times, the duration and the deadline, of every periodic task to be checked for scheduling. This scenario is written into a file which is to be executed by the command interpreter. The simulator is activated by giving the name of this file, the duration between two clock ticks in microseconds, the number of ticks per period of the scenario, and the maximum number of tasks that the simulator will handle.

4.4 *Program Debugging in Real-Time Application Context*

This assistance is provided by the full Unix interface compatibility of Dune_iX. This has been demonstrated by a musical performance, the objective of which was to show a real-time application working concurrently with other applications. (Figure 16)

The demonstration includes:

- A musical synthesizer which is driven by a MacIntosh SE and which produces a musical sound output simultaneously to an audio amplifier and to a Dune 3000 real time board.
- A workstation, which records the activity of the 3 CPUs and displays the ratio of each CPU being either idle or busy with asynchronous (TD) or real time tasks (TI).
- A workstation used to display the musical score which is composed in real-time by the Dune 3000 computer.
- A Dune 3000 computer with 3 CPU, 8 MBytes of common memory, one system I/O board, and one real-time board. The calculation of the musical score, the graphical application enabling display and ordinary Unix file operations are performed on the Dune 3000.

Although the frequency of the musical signal issued from the synthesizer has been multiplied by 4 in order to increase the workload of the CPUs, the musical score is built as the musical sound is heard and is not delayed when file accesses are executed concurrently.

4.5 *Multiprocessing Immediate Tasks Performances*

A Dune 3000 configuration with 4 processors has been interrupted at a frequency of 1000 interrupts/second, through the VME bus. Each interrupt triggers an immediate task of known duration. The measured load of each processor is shown in the table below.

Duration of immediate tasks	Input load of each CPU	Measured load of each CPU	Interrupt latency overhead
50 μs	1.25%	1.5%	20%
100 μs	2.5%	2.6%	4%
1 ms	25%	25.7%	2.8%

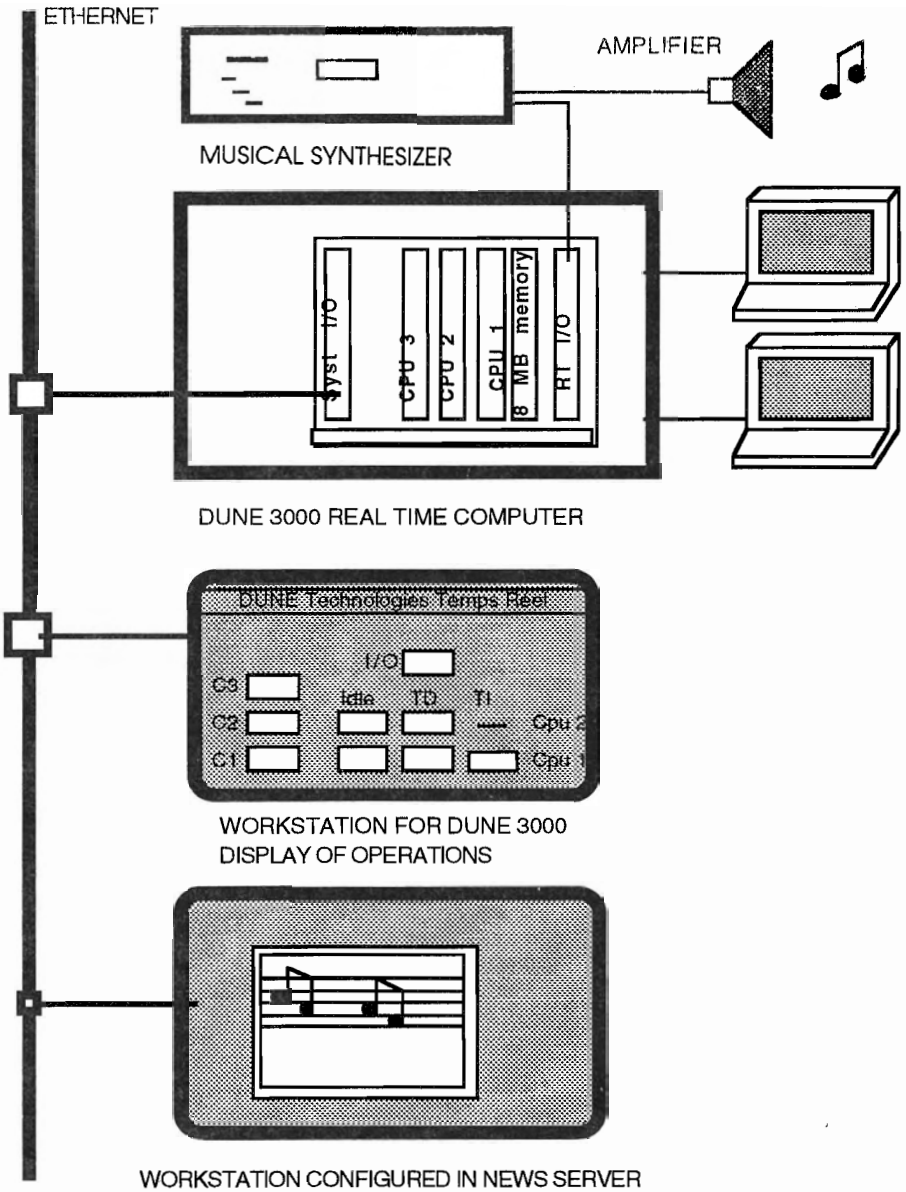


Figure 16. Dune_iX musical demonstration

6. *Dune Who's Who*

The Dune_iX Real-time Operating System was a joint development effort of LETI, a research group of CEA, the French Atomic Energy Research Institution, and of Dune Technologies from October 1989 to June 1991.

The members of the LETI/CEA team were Michel Bastien, Daniel Bras and Jean Delcoigne.

The members of the Dune Technologies team were Jean-Marc Barreteau, Alain Jaouen, Marc Lombard, Albin Pouilles and Gérard Morisset.

The Dune 3000 architecture has been designed and developed by a team of Dune Technologies whose members were Jean Barbier, Olivier Lepape and Frédéric Réblewski.

The Dune_iX project was led by Jean-Serge Banino.

References

[Bauer 1990] R. Bauer. How real is real-time Unix? A review of LynxOs. *Unix Review vol 8, n°9,* pp. 81–88, September 1990

[Bennet 1988] S. Bennet. *Real-Time Computer Control: An Introduction.* Prentice Hall, 362 pages 1988

[Bétourné 1970] C. Bétourné, J. Boulenger, J. Ferrié, C. Kaiser, S. Krakowiak, J. Mossière. Process management and resource sharing in the multiaccess system Esope. *Comm. ACM Vol. 13, 12.* pp. 727–733. Dec. 1970

[Burns 1990] A. Burns, A. Wellings. *Real-time systems and their programming languages.* Addison Wesley, 575 pages, 1990

[Burns 1991] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal, Vol. 6, N 3,* pp. 116–128, May 1991

[Chetto 1989] M. Chetto, H. Chetto. Scheduling periodic and sporadic tasks in a real-time system. *Information Processing Letters, Vol. 30, N°4,* pp. 177–184, Feb. 1989

[Gallmeister 1991] B. Gallmeister. Portable Posix in Real-time. *Unix Review, vol.9, n°4,* pp. 32–36, April 1991.

[Habermann 1976] A. N. Habermann. *Introduction to Operating System Design.* 372 pages. SRA 1976.

[Havender 1968] J. W. Havender. Avoiding Deadlocks in Multitasking Systems *IBM Syst. J. 7.2,* 1968. pp. 74–84.

[Kaiser 1983] C. Kaiser. Mutual exclusion and priority scheduling. *Technology and Science of Informatics 1.1,* 1983. pp. 41–49. North Oxford Academic.

[Lelann 1990] G. Lelann. Critical Issues for the Development of Distributed Real-Time Systems *INRIA. RR 1274.* 1990. 19 pages.

[Levi 1990] S. T. Levi, A. K. Agrawala. *Real-Time System Design.* McGraw-Hill, 299 pages, 1990

[Lister 1984] A. M. Lister. *Fundamentals of Operating Systems.* MacMillan Press. 161 pages. 1984

[Organick 1972] E. I. Organick. *The Multics System: an examination of its structure.* MIT Press 1972

[Posix 1992] Posix 1003.4. Real-time Extension for Portable Operating System. *Posix Technical Committee on Operating Systems of the IEEE Computer Society 1992*

[Sha 1987] L. Sha, R. Rajkumar, J. P. Lehoczky. The Priority Inheritance Protocol: an Approach to Real-Time Synchronization. *CMU-CS 87–181*. 1987.

[Small 1988] C. H. Small. Real-time operating systems. EDN Journal, January 7, 1988. pp 115–138.

[Stankovic 1987] J. Stankovic, K.Ramamritham. The Spring Kernel. *Proc. 8th IEEE Real-Time Systems,* San Jose, California, pp. 146–157, Dec.1987

[Stankovic 1988] J. Stankovic. Misconceptions about Real-Time Computing. *Computer, 21,* October 1988. pp. 10–19

[Tanenbaum 1987] A. Tanenbaum. *Operating System Design and Implementation*. Prentice Hall. 1987

[Wilkes 1970] M.V. Wilkes. *Time Sharing Computer Systems*. Elsevier MacDonald. 102 pages 1970

Illustration 1. Unix and Real-Time Kernels

Adopting Unix means facilities to easily equip a board level system with standard de facto interfaces such as network interfaces or graphical users interface like X-windows, program compatibility and therefore access to Unix packages and tools.

However Unix presents a mix of corporate requirements and technical solutions which reflect the state of the art of the early 70s when it was designed and which don't fit for real-time.

The challenge for real-time standards is between standard non real-time Unixes modified for real-time enhancements and real-time kernels which are standardized by adopting the Unix standard interface.

Standard Unix

Unix developers at Bell Labs of ATT cheerfully admit that they set up Unix for program development to help build research software. The goal was to let programmers be able to work simultaneously on big programming projects. In such an environment, the computing load was not particularly heavy, the response time was at human rate (1/30th of a second) and the Unix tasks requires little intercommunication more than piping an output of a task to the input of another.

Unix designers were uninterested in some prime concern of real-time systems such as deterministic response to interrupt, prioritized and preemptive multitasking scheduler, fast interprocess communication, operating system calls that execute quickly and can be shared among tasks, a secure and fast file system, the ability to recover quickly and safely from outages.

When Unix was first developed, interactive full screen editors were not yet available; programmers used electromechanical teletype machines which were slow and they were motivated to fill a line with as many commands as possible; this gifted Unix with a succinct, cryptic user interface that is both hard to learn and easy to make mistakes with.

The shell program interprets the commands typed by the user and usually creates another task to provide the service requested. The shell

then hangs up, waiting for the end of its child task before continuing with the shell script.

Because of small main memories, standard Unix assumes that the operating system must swap tasks in and out of memory frequently.

The Unix kernel schedules tasks on a modified time-sliced round-robin basis; the priority is ruled by the scheduler and is not defined by the user.

The standard Unix kernel is not particularly interested in interrupts which come usually from a terminal and from memory devices. Data coming into the system does not drive the system as it does in real-time systems. The kernel is, by design, not pre-emptible. Once an application program makes an operating system call, that call runs to completion. As an example of this, when a task is created, by a fork, the data segment of the created task is initialized by copying the data segment of the creator task; this is done within the system call and may last as much as some hundred milliseconds.

Thus all standard Unix I/O is synchronous or blocked and a task cannot issue an I/O request and then continue with other processsing. Instead, the requesting task waits until the I/O call is completed.

A task does not communicate with I/O devices directly and turns the job over to the kernel which may decide to simply store the data in a buffer. Early Unix designers optimized the standard file system for flexibility, not speed, nor security, and consequently highly variable amounts of time may be spent finding a given block of data depending on its position in the file.

Standard Unix does not include much interprocess communication and control. The "pipe" mechanism allows to couple the output of a task to the input of another task of the same family.

The other standard interprocess communication facility is the "signal." The signal works like a software interrupt.

Standard Unix does allow programmers to set up shared memory areas and disk files. Later versions have a slow semaphore mechanism for protecting shared resources.

Standard Unix allows designers to implement their own device drivers and to make them read or write data directly into the memory of a dedicated task. However this is kernel code and the kernel has then to be relinked.

Toward real time Unix

As conventional Unix does not provide adequate response time and data throughput required for supporting real-time applications, many attempts have been made to adapt the Unix kernel to provide a real-time environment.

The real-time enhancements have been sought after in associating a companion real-time kernel and/or improving the standard kernel.

A companion real-time kernel is inserted, along with its associated real-time tasks. It may use a specific processor. It functions apart of the Unix kernel. It is in charge of the reactions to interrupts and schedules as many real-time tasks as necessary for these reactions. To allow this, the Unix kernel is preempted by its companion kernel. However when some real-time data have to be forwarded to the Unix programs, this communication between the companion kernel and Unix is always done in a loosely coupled mode and the transfer has to be finalized in the Unix program; the non-deterministic Unix scheduler wakes up the application program and therefore there is no real-time behaviour.

Some Unix kernels have been reworked to improve their real-time performances. As the basic kernel is not preemptive, it can only be split into processing steps that must run to completion without being interrupted. In between these processing steps, preemption points are identified where the kernel can safely interrupt its processing and schedule a new task.

Besides these limited response time ameliorations, additional features need to be provided for lower kernel latency, such as kernel-level, preemptable tasks called daemons, locking a task and its segments in main memory, locking pages in memory or reserving access to a bus for a specific processs, additional priority levels, modified schedulers, autonomous system traps, direct communication between I/O device and a task, contiguous files, faster file indexing schemes, named pipes, event mechanisms, gang scheduling . . .

Extending Unix to real-time requires also to extend the Unix interface. These efforts should result in the definition of the POSIX IEEE 1003.4 standards.

Real Time Kernels Opening to the Unix World

A customized real-time kernel replaces completely the Unix kernel by another kernel which provides a real-time interface and a standard interface. The basic idea is that real-time applications don't need the Unix system or kernel but require Unix interfaces.

These kernels have native real-time nucleus, which present usual real-time capabilities. Their basic interface has been augmented with a full Unix interface providing source or binary compatibility for existing Unix programs. Thus their interface is a superset of the Unix interface.

Real-time Performances of Unix-like Approaches

An appraisal of these systems which attempt to provide a standardized real-time kernel may rely on their capability to manage interrupts, to limit the clerical latency caused by the transfer of real-time data from the interrupting device to the application Unix tasks, to insert interrupt routines for application needs.

The standard Unix kernel:

- preempts the current task to notice only that an interrupt requires to process an answer when possible,
- calls the task scheduler at the end of time slices only, awaking the required task according to a time-sharing scheduling policy,
- inserts interrupt routines only in the kernel space. The clerical latency may be up to several 100 milliseconds.

A companion real-time kernel:

- allows to handle interrupts and to acquire data in real-time,
- however data transfer remains under the Unix kernel control.

Therefore the clerical latency remains up to some 10 milliseconds.

A reworked Unix kernel can improve the clerical latency down to about a few milliseconds.

Only preemptive and fully reentrant kernels can switch their context at any time and instantaneously react to interrupts. For example,

the VRTX real-time kernel has an interrupt latency of about 10 microseconds.

This leads to a clerical latency of 500 microseconds.

When, in addition, interrupt routines can be written in the user space, data transfer is suppressed and additional context switches are avoided, which reduces the clerical latency to 300 microseconds.

The above figures on latencies assume a processor such as a Motorola 68030 (at 25 Mhz).

Illustration 2. Multiprocessor Optimized Spin Lock

```
procedure mit() is
begin
   mask_interrupts;
   go := test_and_set(m); --m and go are boolean
   while not go loop          --global busy waiting
     unmask_interrupts;
     while not m loop null; endloop;    --local busy waiting
     -- no bus access contention, only local cache accesses
     mask_interrupts;
     go := test_and_set(m);
   endloop;
 end mit();

procedure rit()is
begin
   m := true; --leaving critical section
   unmask_interrupts;
end rit();

begin
  mit();    --entering kernel critical section
  shared_kernel_data_mutually_exclusive_utilisation;
                  --note that interrupts are masked
  rit();
end of example;
```

Notice that the masking of interrupts is necessary for avoiding deadlock.

Illustration 3. Multiprocessor Optimized Semaphores

```
 generic
        INITIAL: NATURAL := 1; --default value is 1
package SEMAPHORE is
       procedure P;
       procedure V;
end SEMAPHORE;

package body SEMAPHORE is
       SEMA_LOCK: BOOLEAN := TRUE;
                -- a test_and_set will set it to FALSE
       COUNT: NATURAL:= INITIAL;
       QUEUE: TASK_QUEUE;

procedure P is
GO, TO_BE_SWITCHED: BOOLEAN;
 begin
    TO_BE_SWITCHED := FALSE;
    INVCOM; --this procedure invalidates_task_switching
    GO := test_and_set(SEMA_LOCK);
    while not GO loop      --global busy waiting
      VALCOM; --validates_task_switching;
      while not SEMA_LOCK loop null; endloop;
      -- local busy waiting
      -- no bus access contention, only local cache accesses
      INVCOM; --invalidates_task_switching;
      GO := test_and set(SEMA_LOCK);
    endloop;
    COUNT := COUNT - 1;
    if COUNT < 0 then
          QUEUE.ENTER(CURRENT_TASK);
          TO_BE_SWITCHED := TRUE;
   endif;
  SEMA_LOCK := TRUE;
  VALCOM; --validate_task_switching;
  if TO_BE_SWITCHED then SCHEDULE_AND_SWITCH; endif;
end P;
```

```
procedure V is
   GO, TO_BE_SWITCHED: BOOLEAN;
begin
   TO_BE_SWITCHED := FALSE;
   INVCOM; --invalidates_task_switching;
   GO := test_and_set(SEMA_LOCK);
   while not GO loop          --global busy waiting
      VALCOM; --validates_task_switching;-
      while not SEMA_LOCK loop null; endloop;
   --local busy waiting
   --no bus access contention, only local cache accesses
      INVCOM; --invalidates_task_switching
      GO := test_and set( SEMA_LOCK );
   endloop;
   COUNT := COUNT + 1;
   if COUNT <= 0 then QUEUE.EXTRACT(A_TASK);
            TO_BE_SWITCHED := TRUE;
   endif;
   SEMA_LOCK := TRUE;
   VALCOM; --validates_task_switching
   if TO_BE_SWITCHED then SCHEDULE_AND_SWITCH; endif;
end V;
end package body SEMAPHORE;
```

[submitted Dec. 12, 1992; revised April 1, 1993; accepted Apr. 8, 1993]