THE DESIGN OF THE SPRING KERNEL

John A. Stankovic and Krithi Ramamritham
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 88-85

The Design of the Spring Kernel*

John A. Stankovic
Krithi Ramamritham

Technical Report 88-85

Dept. of Computer and Information Science
University of Massachusetts
Amherst, Mass. 01003

October 3, 1988

Abstract

Next generation real-time systems will require greater flexibility and predictability than is commonly found in today's systems. These future systems include the space station, integrated vision/robotics/AI systems, collections of humans/robots coordinating to achieve common objectives (usually in hazardous environments such as undersea exploration or chemical plants), and various command and control applications. The Spring kernel is a research oriented kernel designed to form the basis of a flexible, hard real-time operating system for such applications. Our approach provides a method for on-line dynamic guarantees of deadlines. This provides many benefits. The main contributions in our approach are the scheduling algorithms themselves, the design of the kernel that enables predictability of execution time, and the synergism between the scheduling algorithm and the kernel design. The Spring kernel is being implemented on a network of (68020 based) multiprocessors called SpringNet.

*This work was supported by ONR under contracts NOOO14-85-K-0389 and NSF under grant DCR-8500332. A short version of this paper appeared in Proceedings of the Real-Time Systems Symposium, 1987.

1 Introduction

Recently, there has been an increased interest in hard real-time systems and such systems are becoming more and more sophisticated. We define Hard Real-time systems as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. Further, if these real-time constraints are not met there may potentially be catastrophic consequences. Examples of this type of real-time system are command and control systems, nuclear power plants [1], process control systems, flight control systems, and the space shuttle avionics system [7]. In the future, such systems are expected to become more and more complex, have long lifetimes, and exhibit very dynamic, adaptive and even intelligent behavior. These future systems include the space station, integrated vision/robotics/AI systems, collections of humans/robots coordinating to achieve common objectives (usually in hazardous environments such as undersea exploration or chemical plants), and various command and control applications.

The most critical part of supporting such new systems is the ability to guarantee that real-time constraints can be met. Because of the large number of combinations of tasks that might be active at the same time and because of the continually changing demands on the system, it will generally be impossible to pre-calculate all possible schedules *off-line* to statically guarantee real-time constraints. Our approach is to perform research on providing a method for on-line dynamic guarantee of deadlines. This approach allows the unique abstraction that at any point in time the operating system knows exactly what set of tasks are guaranteed to make their deadlines, what, where and when spare resources exist or will exist, and which tasks are running under non-guaranteed assumptions. The keys to our approach are the scheduling algorithm, the operating system, and their synergism.

Real-time systems usually include a real-time kernel [9] [24]. However, most existing real-time kernels [28] are simply stripped down and optimized versions of timesharing operating systems. More specifically, the general characteristics of most *current* real-time kernels typically include:

- a fast context switch,
- a small size (with its associated minimal functionality),
- the ability to respond to external interrupts quickly,
- multi-tasking with task coordination being supported by features such as ports, events, signals, and semaphores,

- fixed or variable sized partitions for memory management (no virtual memory),
- the presence of special sequential files that can accumulate data at a fast rate,
- priority scheduling,
- the minimization of intervals during which interrupts are disabled,
- support of a real-time clock,
- primitives to delay tasks for a fixed amount of time and to pause/resume tasks, and
- special alarms and timeouts.

These features provide a basis for a good set of primitives upon which to build real-time systems. These features are also designed to be fast which is a laudable goal. However, fast is a relative term and not sufficient when dealing with real-time constraints. The main problems with these primitives are that they do not *explicitly* address real-time constraints, nor does their use (without extensive simulations) provide system designers with a high degree of confidence that the system will indeed meet its real-time constraints. Even though such kernels are successfully used in today's real-time embedded systems, it is only at extremely high cost and inflexibility. For example, when using the above primitives it is difficult to *predict* how tasks invoked dynamically interact with other active tasks, where blocking over resources will occur, and what the subsequent effect of this interaction and blocking is on the timing constraints of all the tasks. The current technology burdens the designer with the unenviable task of mapping a set of specified real-time constraints into a priority order in such a manner that all tasks will meet their deadlines. It is common practice to attempt to verify real-time constraints under such conditions by extensive and costly simulations and testing on the actual system [10]. One round of changes is subject to another *extensive* round of testing. As the next generation hard real-time systems become more sophisticated, it will be necessary to develop cheaper ways to guarantee real-time constraints and to meet the flexibility requirements [30] [29]. The main characteristics of *next* generation hard real-time systems are:

- new operating system and task designs to support predictability,
- a high degree of adaptability (short term and long term),
- physical distribution (of multiprocessors) with a high degree of cooperation,
- incorporation of integrated solutions to deal with real-time, fault tolerance, and large system requirements,
- an interface to AI programs,

- the ability to handle complex applications,
- the ability to integrate cpu scheduling with resource allocation, and
- the ability to determine end-to-end timing performance, e.g., if task A sends a message to task B under timing constraints then all aspects of the communication must be accounted for including context switching, message transmission and reception, and remote site scheduling delay.

The Spring project at the University of Massachusetts is conducting research into next generation hard real-time systems. The project has many thrusts [21]. The three main thrusts that directly relate to the Spring kernel are:

1. The development of dynamic, distributed, on-line real-time scheduling algorithms. This work is well underway with many such algorithms already developed and evaluated, each based on a different set of assumptions. The plan is that any of these algorithms can plug into the kernel depending on the requirements and policies of the application.
2. The development and implementation of the Spring kernel which supports a network of multiprocessors. The design of the kernel is now complete and a plan for rapid prototyping the kernel has been established. The purpose of this paper is to describe the main ideas in the design of the Spring kernel. The major innovations exhibited in the Spring kernel are the scheduling algorithm itself which can dynamically guarantee real-time constraints, and the way in which the rest of the kernel supports the scheduling algorithm. For example, one major feature is that the scheduling algorithm and other primitives of the kernel cooperate to *avoid* blocking, thereby making it possible to attain predictability.
3. The development of multiprocessor nodes in order to directly support the kernel and the scheduling algorithm. The multiprocessor systems are Motorola Systems 1131s and 1132s based on the 68020 processors and the VME bus.

The remainder of this paper is organized as follows. Section 2 introduces our model of a hard real-time system. Section 3 describes one scheduling algorithm. This scheduling algorithm avoids unpredictable waiting and accounts for the use of exclusive and shared resources. Other versions of the algorithm are briefly discussed. Section 4 describes the main primitives found in the Spring kernel. This includes the task management primitives, the memory management primitives, the IPC primitives, and a discussion of the I/O subsystem and interrupt handling. The Spring kernel does not contain support for security, multiple address spaces, general timesharing (although it does support the concurrent existence of hard- and soft- and

non- real-time tasks), nor is it intended for development into a production operating system with all of the incumbent overheads and size. It is a research vehicle into developing flexible, predictable, next generation hard real–time systems. Section 5 discusses a number of miscellaneous issues with respect to the kernel design and implementation. Finally, a summary is presented in Section 6.

2 System Model

This section presents the basic structure of a distributed real-time system that we are assuming. It is based on the notion of a flexible on-line scheduler that can guarantee that tasks make their deadlines [18]. While the details of our scheduling algorithm and the analysis of them have appeared elsewhere [18], [26], [31], we will repeat the basic ideas of the algorithm in this paper with the intent of showing how it interfaces to the rest of the Spring kernel, and why it is different than today's real-time kernels. This scheduling algorithm is, in fact, a major component of the kernel.

We assume that the Spring system is physically distributed and composed of a network of multiprocessors. See Figure 1. Each multiprocessor contains one (or more) application processors, one (or more) system processors, and an I/O subsystem. All processors are 68020s and all processors have their own local memory. All processors and memories are attached to a VME bus, forming a single global memory space per node. System processors[1] offload the scheduling algorithm and other OS overhead from the application tasks both for speed, and **so that this overhead does not cause uncertainty in executing guaranteed tasks.** All system tasks are resident in the memory of the system processors. The I/O subsystem is a separate entity from the Spring kernel and it handles non-critical I/O, slow I/O devices, and fast sensors. The I/O subsystem can be controlled by some current real-time kernel such as VRTX [22], or by completely dedicating processors or cycles on processors to these devices. The I/O subsystem interface to the Spring kernel is best explained after the scheduling algorithm is described so we defer any more discussion of I/O until section 4.4.

It is important to note that although system tasks run on system processors, application tasks can run on both application processors and system processors by explicitly reserving time on the system processors. This only becomes necessary if the surplus processing power of the application processor(s) is (are) not sufficient at a given point in time. If both the application processors and a portion of the system processors are still not sufficient to handle the current load, then we invoke

[1] Ultimately, system processors could be specifically designed to offer harware support to our system activities such as guaranteeing tasks.

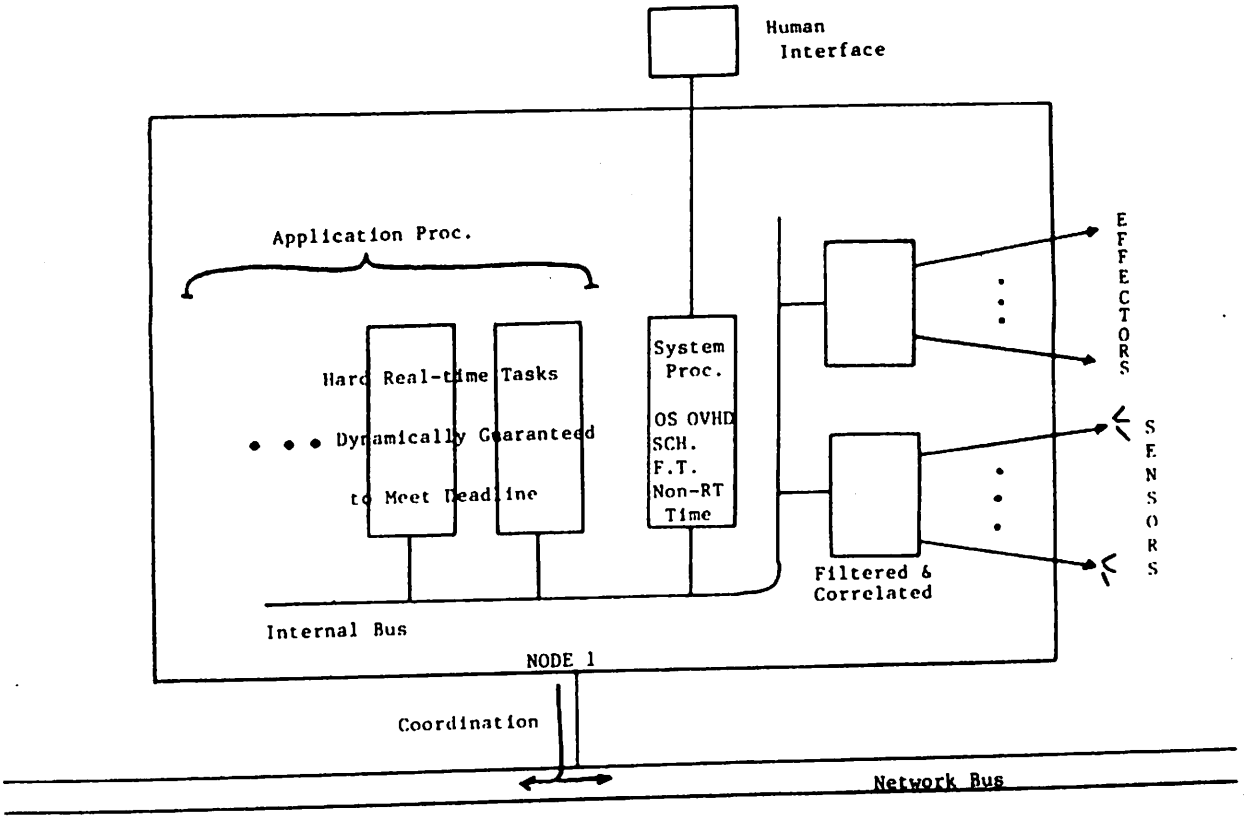


Figure 1: A Spring Node

the distributed scheduling portion[2] of our algorithm. Some modifications to our previously reported work have been made for implementing distributed scheduling on SpringNet. Most notably, the code for tasks is now replicated at various nodes, so that only signals, partial state information, or input to the tasks need be transmitted when distributed scheduling occurs, rather than transmitting the task code itself.

To be more specific, the system processors run most of the operating system, as well as application specific tasks that do not have deadlines. The scheduling algorithm separates policy from mechanism and is composed of 4 modules, one of which can be used in two different ways. At the lowest level multiple dispatchers exist, one running on each of the application processors. The dispatcher simply removes the next (ready) task from a system task table (STT) that contains all guaranteed tasks arranged in the proper order for each application processor. The rest of the scheduling modules are executed on the system processor. The second module is a local scheduler. The local scheduler can be used in two ways. First, the local scheduler is responsible for locally *guaranteeing* that a new task can make its deadline, and for ordering the tasks properly in the STT. The logic involved in this algorithm is a major inovation of our work. Second, the local scheduler can also be invoked as a *time* planner – valuable for real-time AI applications. This important idea means that it is possible to consider the impact of system level allocations and resource conflicts on the execution time properties of application tasks and that this information can then be used by the application to more accurately accomplish goals on time. Using the local scheduler as a planner is considered a high level OS activity and therefore will not be discussed any further in this paper. The third scheduling module is the global (distributed) scheduler which attempts to find a site for execution for any task that cannot be locally guaranteed. The final module is a Meta Level Controller (MLC) which has the responsibility of adapting various parameters by noticing significant changes in the environment and serving as the user interface. The distributed scheduling component and the MLC are not discussed any further in this paper since they can be considered upper levels of the OS and are not part of the Spring kernel itself.

2.1 Tasks

At the kernel level there exists an executable and guaranteeable entity called a task. A task consists of reentrant code, local data, dynamic data segments, a stack, a task descriptor and a task control block. Multiple instances of a task may be invoked. In this case the reentrant code and task descriptor are shared.

Tasks are characterized by:

[2]See [18] [19] [20] for details on distributed scheduling.

- ID
- Group ID, if any (tasks may be part of a task group or a dependent task group - these are more fully explained below)
- C (a worse case execution time) (may be a formula that depends on various input data and/or state information)
- Deadline (D) or period or other real–time constraints
- criticalness (this is an indication of the importance of this task)
- preemptive or non-preemptive property
- maximum number and type of resources (this includes memory segments, ports, etc.) needed
- type: non RT, soft RT, or hard RT
- incremental task or not (incremental tasks compute an answer immediately and then continue to refine the answer for the rest of its requested computation time)
- precedence graph (describes the required precedence among tasks in a task group or a dependent task group)
- communication graph (list of tasks with which a task communicates), and type of communication (asyn or syn)
- location of task copies

All the above information concerning a task is maintained in the task descriptor. Much of the information is also maintained in the task control block with the difference being that the information in the task control block is specific to a particular instance of the task. For example, a task descriptor might indicate that the worst case execution time for TASK A is 5z milliseconds where z is the number of input data items at the time the task is invoked. At invocation time a short procedure is executed to compute the actual worst case time for this module and this value is then inserted into the TCB. The guarantee is then performed against this specific task instance. All the other fields dealing with time, computation, resources or criticalness are handled in a similar way.

While the kernel supports tasks, the local scheduler not only guarantees with respect to tasks, but also supports the abstractions of task groups and dependent task groups. A task group is a collection of simple tasks that have precedence constraints among themselves, but have a single deadline. Each task acquires resources before

it begins and can release the resources upon its completion. For task groups, it is assumed that when the task group is invoked, all tasks in the group can be sized (this means that the worst case computation time and resource requirements of each task can be determined at invocation time). A dependent task group is the same as a task group except that only those tasks with no precedence constraints can be sized at invocation time. The remaining tasks of the dependent group can only be sized when all preceding tasks are completed. The dependent task group requires some special handling with respect to guarantees. Our work in this area is tentative and hence is not discussed further in this paper.

Note that a simple task may be guaranteed to execute as a simple task, and also as part of some group. Tasks in a task group can communicate via shared memory or the IPC primitives.

2.2 Principle of Segmentation

The design of the kernel is based on the principle of segmentation as applied to hard real-time systems. Due to space limitations we cannot fully discuss the use of segmentation and its implications in this paper. Further details on segmentation for hard real-time systems can be found in [27]. We present a brief description of the segmentation principle here to provide some motivation for our kernel design decisions. Segmentation is a key idea in being able to guarantee end-to-end timing constraints and to integrate cpu scheduling and resource allocation.

Segmentation is the process of dividing resources of the system into units where the size of the unit is based on various criteria particular to the resource under consideration and to the application requirements. For example, dividing time on a bus into fixed slots with fixed start times is an example of segmentation. Dividing time on a bus into slots of two different sizes is also segmentation which might be more suitable for application environments where there is roughly a bimodal distribution of message sizes. Dividing time into bounded size packets, but allowing the start of the packet to begin anywhere is yet another "more relaxed" example of segmentation in that the packet size is segmented, but the bus time is not. Allowing variable length messages which can begin at any time is not segmentation. The goals of using segmentation in hard real-time systems are to develop well defined units of each resource, to increase understandability, and to allow an on-line algorithm to assemble the units in a way that provides predictability with respect to timing constraints.

Segmentation is a powerful concept and is used in many circumstances. For example, timesharing systems relate the size of a page in memory with block sizes on disks. One can refer to this as spatial segmentation. Hardware designers strive to

balance the data flow cycle time and data path widths with the timing of the control memory, local store, and main memory so that no one component is either overdesigned or is a bottleneck to performance. Thus hardware designers are integrating spatial (data path widths) and multiple timing segments with respect to the hardware. While segmentation is used in some form in many systems, we are advocating the need to elevate the notion of segmentation to a central *principle* of hard real-time systems. That is, we must extend the integration of spatial and timing segmentation to the system level, not just the hardware level.

For example, in a hard real-time system *time* is one of the most important resources to segment. However, a system is composed of many different time segments, e.g., time segments for the cpu, for the network bus, for the internal bus, for the disk controller, etc. The system is being driven by multiple, coordinated drum beats (i.e., time segments) with small differences of time granularity being managed by various techniques such as latches[3]. Device virtualization is used when timing differences are greater. For example, memory might be used as a virtual disk, so that time to store data on this virtual disk is fast and predictable, and the actual disk write is done in background mode without a severe timing constraint. Again, one way of thinking of time segmentation is as if the timing and control circuits of a cpu are being extended to the entire system. But timing segmentation is not sufficient by itself. Other resources such as data and memory are segmented with respect to functionality and size, and programs are segmented with respect to functionality, size and time. For example, the following types of segments exist: code, task control blocks (TCB), task descriptors (TD), local data, data (including shared memory), ports, stacks, virtual disks, and non segmented memory. TCBs, TDs, virtual disks, and stacks are of fixed size while the other types of segments have variable size, but the size is fixed at invocation time. By design, code segments are not large and have small variance in execution time, if at all feasible.

3 The Scheduling Algorithm

The scheduling algorithm has several primary contributions including: the ability to perform a guarantee on-line, the ability to utilize all the nodes of a distributed system for a hard real-time system, the ability to predict when a task or set of tasks cannot meet their deadline, resulting in the feature of being able to predict timing faults before they occur. The main ingredient of the scheduling algorithm is the guarantee routine.

[3]Latches are flip-flops organized as a storage register used to hold signals for brief periods to overcome small differences in timing between the cpu and other system components.

The basic notion and properties of guarantee have been developed elsewhere [18] and have the following characteristics,

- it integrates cpu scheduling with resource allocation,
- conflicts over resources are *avoided* thereby eliminating the random nature of waiting for resources found in timesharing operating systems (this same feature also tends to minimize context switches since tasks are not being context switched to wait for resources),
- there is a separation of dispatching and guarantee allowing these system functions to run in parallel; the dispatcher is always working with a set of tasks which have been previously validated to make their deadlines and the guarantee routine operates on the current set of guaranteed tasks plus any newly invoked tasks,
- early notification: by performing the guarantee calculation when a task arrives there may be time to reallocate the task on another host of the system via the global module of the scheduling algorithm; early notification also has fault tolerance implications in that it is now possible to run alternative error handling tasks early, before a deadline is missed,
- the guarantee can employ different strategies for deciding if a task can meet its deadline as a function of the deadline, resource requirements, criticalness, and precedence constraints of the incoming task,
- using segments and precedence constraints it is possible to guarantee end-to-end timing constraints,
- within this approach there is notion of still "possibly" making the deadline even if the task is not guaranteed, that is, if a task is not guaranteed it receives any idle cycles and in parallel there is an attempt to get the task guaranteed on another host of the system subject to location dependent constraints,
- some real-time systems assign fixed size slots to tasks based on their worst case execution times, we guarantee based on worst case times but any unused cpu cycles are reclaimed when resource conflicts don't prohibit this reclamation and not lost,
- worst case execution time is computed for a specific invocation of a task and hence will be less pessimistic than the absolute worst case execution time,
- the guarantee routine supports the co-existence of hard and soft real-time tasks, and

- the guarantee can be subject to computation time requirements, deadline or periodic time constraints, resource requirements where resources are segmented, criticalness levels for tasks, precedence constraints, I/O requirements, etc. depending on the specific guarantee algorithm in use in a given system. This is a realistic set of requirements. Note that current real-time executives provide little support with respect to handling tasks with deadlines and general resource requirements, and no support for end-to-end scheduling.

The Segmentation principle applied to the kernel provides well defined resource units. The guarantee algorithm maps the task requirements onto the resource segments. In general, this mapping is NP–hard, hence heuristics are required. We now describe the scheduling algorithm and give an example to provide the reader a full understanding of the problem.

The Spring kernel local scheduler considers the problem of scheduling a set of n tasks $\mathcal{T}$, in a system with r resources $\mathcal{R}$. To simplify the discussion, we first describe the algorithm for independent tasks on nodes with a single application processor and a single system processor. At the end of this section we then describe the extensions needed to handle precedence constraints, periodic tasks, multiple application processors and criticalness. We will also discuss the run time costs of the algorithm.

We now begin the discussion by concentrating on the most difficult aspect of scheduling, handling the resource requirements of tasks. It is this aspect of scheduling that provides resource conflict avoidance and thereby predictability.

A resource can be used in two different modes: When in *shared mode*, several tasks can use the resource simultaneously; when in *exclusive mode*, only one task can use it at a time. A file or data structure are examples of such resources: a file can be *read* by multiple users simultaneously but can be *written* by a single user only. A CPU, on the other hand, is a resource that can be used only in exclusive mode. Each task $T \in \mathcal{T}$, has

1. *Processing time,* $T_P > 0$,
2. *Deadline,* T_D,
3. *Resource requirements,* $\mathbf{T_R} = (T_R(1), T_R(2), ..., T_R(r))$, where

$$T_R(i) = \begin{cases} 0 & \text{T does not require resource } R_i; \\ 1 & \text{T requires } R_i \text{ in shared mode;} \\ 2 & \text{T requires } R_i \text{ in exclusive mode,} \end{cases}$$

and

4. *Scheduled start time,* T_{sst} (determined from the processing time, deadline, and resource requirements of all tasks).

A *partial schedule* is a subset of the tasks in $\mathcal{T}$ whose scheduled start times have been assigned. A partial schedule S is *feasible* if the scheduled start times are such that all the tasks in S will meet their deadlines, i.e., $\forall T \in S, (T_{sst} + T_P \leq T_D)$. For the tasks in a feasible schedule, the resources required by each task are available in the mode required by the task at its scheduled start time. A set of tasks is *schedulable* if there exists a feasible schedule for it. Thus, the scheduler must determine if a feasible schedule for a set of tasks exists. Also, it should be obvious from the above description that we are interested in non-preemptive scheduling. Thus, once a task begins execution, it will release its resources only after it has executed for T_P units of time.

Suppose tasks in set Γ have been previously scheduled and a new task arrives. We attempt to schedule the set of tasks $\Pi = \Gamma \bigcup \{\text{new task}\}$. If this set of tasks is found schedulable, the new task is scheduled, otherwise not. In either case, tasks in Γ remain scheduled.

For a given set of tasks, the problem of finding a feasible schedule is, in fact, a search problem. The structure of the search space is a *search tree*. The *root* of the search tree is the empty schedule. An *intermediate vertex* of the search tree is a partial schedule. A *leaf*, a terminal vertex, is a complete schedule. Note that not all leaves correspond to feasible schedules. The goal of the scheduling algorithm is to search for a leaf that corresponds to a feasible schedule.

An optimal algorithm, in the worst case, may make an exhaustive search which is computationally intractable. In order to make the algorithm computationally tractable even in the worst case, we take a heuristic approach for this search. We develop a heuristic function, H. That is, on each level of the search, function H is applied to each of the tasks that remain to be scheduled. The task with the minimum value of function H is selected to extend the current (partial) schedule. As a result of the above directed search, even in the worst case, our scheduling algorithm is not exponential. Fortunately, our simulation studies, described in [31], [32], and [33] show that algorithms using linear combinations of simple heuristics perform very well — very close to the optimal algorithm that uses exhaustive search.

The pseudo code for our scheduling algorithm is given in Figure 2. The algorithm maintains two vectors EAT^s and EAT^e, each element of the vector corresponding to a resource. EAT^s and EAT^e, respectively, indicate the earliest available times of resources in shared and exclusive mode, given that the tasks in *schedule* have been scheduled and tasks in the *task_set* remain to be scheduled. At each level of search,

```
Procedure Scheduler(task_set: task_set_type; var schedule: schedule_type; var
schedulable: boolean);

(*parameter task_set is the given set of tasks to be scheduled*)

VAR EAT^s, EAT^e: vector_type; (* Earliest Available Times of Resources *)

BEGIN

 schedule := empty;

 schedulable := true;

 EAT^e := 0;  (* a zero vector*)

 EAT^s := 0;

 WHILE (NOT empty(task_set)) AND (schedulable) DO

 BEGIN

   calculate T_est for each task T ∈ task_set;

   IF NOT strongly-feasible(task_set, schedule) THEN

     schedulable := false;

     ELSE

     BEGIN

       apply function H to each task in task_set;

       let T be the task with the minimum value of function H;

       T_sst := T_est;

       task_set := task_set - T ;

       schedule := append(schedule, T); (* append T to schedule *)

       calculate new values of EAT^s and EAT^e;

     END;

 END;

END;
```

Figure 2: Heuristic Scheduling Algorithm

according to the earliest available times of resources EAT^s and EAT^e, the algorithm calculates the earliest start time T_{est} for each task which remains to be scheduled. The detailed computation methods for EAT^s, EAT^e, and T_{est} are not discussed here (see [28]).

The algorithm invokes a boolean function called *strongly-feasible*. A feasible partial schedule is said to be strongly-feasible if all schedules obtained by extending this schedule one more level with any one of the remaining tasks are also feasible. If extending a feasible partial schedule by any one of the remaining tasks makes the extended schedule infeasible, then in none of the possible future extensions will this task meet its deadline. Hence it is appropriate to stop the search when a partial schedule is not strongly-feasible.

From the pseudo-code, we see that beginning with the empty schedule, the algorithm searches the next level by expanding the current vertex (a partial schedule) to *only* one of its immediate descendants. If the new partial schedule is strongly-feasible, the search continues until a full feasible schedule is met. At this point, the searching process (i.e., the scheduling process) succeeds and the task set is known to be schedulable.

If at any level, a schedule that is not strongly-feasible is met, the algorithm stops the searching (scheduling) process and announces that this set of tasks is not schedulable and typically either an error message is sent, an error handler is executed, or distributed scheduling is invoked. On the other hand it is also possible to extend the algorithm to continue the search even after a failure, for example, by limited backtracking. While we do not discuss backtracking in detail, we will later present some performance results where we allow some limited amount of backtracking.

Let us now consider some scheduling examples to clarify both the algorithm and the above terminology. Assume that we are attempting to schedule a set of 5 tasks having parameters as shown in Table 1.

If the scheduling algorithm uses an H function defined as $H(T) = T_P$, where T_P is the processing time, we have

$$H(T_2) < H(T_i)$$

for all i = 1, 3, 4, and 5. Hence, the algorithm will select T_2 to to be scheduled first. This new schedule is strongly-feasible and hence the scheduling process will continue with the selection of T_3. This partial schedule is not strongly-feasible since T_1 will miss its deadline. Hence the scheduling process stops immediately.

If the scheduling algorithm uses $H(T) = T_D$, then

$$H(T_1) < H(T_i)$$

Tasks	Processing Time	Deadline	Resource Requirements		
			R_1	R_2	R_3
T_1	20	30	2	2	1
T_2	10	90	2	0	2
T_3	15	40	2	1	0
T_4	20	55	0	2	2
T_5	20	65	0	0	1

Table 1: Parameters of 5 Tasks

for all i = 2, 3, 4, and 5. Hence, T_1 will be selected, followed by T_3 and T_4 at subsequent levels. However, at this point the partial schedule is not strongly-feasible since T_5 will miss the deadline. A smarter scheduler, after selecting T_1, would schedule T_5 next, because T_1 and T_5 can run in parallel. Then by scheduling T_3, T_4 and T_2 in this order, every task will finish before its deadline.

Clearly, at each level of search, effectively and correctly selecting the immediate descendant is difficult, but very important for the success of the algorithm. The heuristic function H becomes the core of the algorithm.

From extensive simulations reported in [33] we have determined that a combination of two factors is an excellent heuristic function H. Consider:

- Min_D + Min_S:

$$H(T) = T_D + W * T_{est};$$

In the above formula, W is a weight, and may be adjusted for different application environments. We have shown that no single heuristic performs satisfactorily and that the above combination of factors does perform well. These two factors address the deadline, the worst case computation time and resource contention – three important issues.

One important aspect of this study, different from previous work, is that we specifically consider resource requirements and model resource use in two modes: exclusive mode and shared mode. We have shown that by modeling two access modes, more task sets are schedulable than if only exclusive mode were used. Further, this algorithm takes realistic resource requirements into account, and it has the appealing property that it avoids conflicts (thereby avoiding waits) over resources. It is important to note that resource conflicts are solved by scheduling at different times tasks which contend for a given resource. This avoids locking and its consequent unknown

delays. If task A is composed of multiple task segments, and task A needs to hold a serially shareable resource across multiple task segments, then that resource is dedicated to task A during that entire period, call it X, and other tasks which need that resource cannot be scheduled to overlap with task A during period X[4]. Other tasks can overlap with task A. This strategy also minimizes context switches, since tasks are not subject to arbitrary interrupts generated by tasks arbitrarily waiting for resources. As an aside, if a particular hard real-time system has no conflict over resources except the CPU then it is possible to assume that resources are always available to ready tasks and one may use our preemptive algorithm [32], instead of the non-preemptive algorithm presented in this paper.

Run time cost of the non-preemptive scheduling algorithm is an important factor to consider. We measured the execution time cost for our algorithm on a VAX 11/780 for task sets of different sizes and for 7 critical resources[5] at each site. The cost of the algorithm depends on the number of backtracks allowed. (In applying the H function, we have capped the number of backtracks.) For example, for 10 tasks and no backtracks the algorithm runs in 22.5 ms. For 10 tasks with with a maximum of n^2 backtracks, the algorithm runs in 51.6 ms. For 30 tasks with backtrack we measured the algorithm as needing 160 ms. These figures are presented to give the reader a feeling for the cost of the algorithm if implemented in software. Hence, without further optimizations nor specialized hardware support for the scheduling algorithm, we expect that the Spring kernel is usually dealing with tasks with laxities of 200 ms or greater. However, we believe that significant optimizations on the algorithm are possible, and have preliminary results in this area. For example, suppose that there are hundreds of active tasks at a given site. In this case we can cap the cost of the algorithm by taking a slice of, say at most 30 tasks, from the STT and trying to reorder just those 30, rather than the entire table. Other optimizations are also possible.

It is also important to note that the execution time cost of the algorithm is a function of the average number of resources that *each* task requires, and not the total number of resources. Consequently, even if there are 100's of critical resources defined per site, as long as each task requires some small number of them (e.g., less than 10), then the impact of the number of resources on the execution time of the algorithm is negligible.

Many extensions to the algorithm described above are possible: First of all, it is easy to immediately extend the algorithm to handle the case where each resource may have multiple instances and a task may request one or more instances of a resource.

[4]General real–time system design rules encourage a programming style in which no task holds a resource for a long period of time (over many segments).

[5]Many other resources may exist, but if they are not shared in some way then they need not be addressed by the guarantee routine.

For this case, the vectors EAT^s and EAT^e will be matrices, each row corresponding to a resource, and each matrix element corresponding to an instance of a resource. Hence, handling multiple application processors is simple, and is accomplished by making the exclusive resource entry for the processor, a vector.

Second, the algorithm can be extended to handle the case where tasks can be started only after some time in the future. For example, this occurs for periodic tasks, and for non-periodic tasks with future start times. Conceptually, the only modification that needs to be made to our scheme is in the definition of tasks' scheduled start time:

$$T_{est} = Max(\text{T's start time}, EAT_i^u)$$

where u = s or e if T needs R_i in shared or exclusive mode, respectively. However, more efficient techniques to handle periodic tasks are being investigated. These techniques are based on a guaranteed template so that each instance of a periodic task need not be guaranteed separately.

Third, in order to handle precedence constraints we simply add another factor to the heuristic function that biases those eligible tasks with long critical paths to be chosen next. A task becomes eligible to execute only when all of its ancestors are scheduled. Precedence constraints are used to model end-to-end timing constraints [8]. Again, various optimizations are being investigated here.

Fourth, a major advantage of our approach is that we can separate deadlines from criticalness. However, for ease of explanation and to emphasize the *avoidance* and resource requirements aspects of our scheduling approach, the above algorithm is described using deadlines only. In actuality, in the first phase the guarantee is performed as described above using deadlines and resource constraints. If the task is guaranteed then the criticalness value plays no part. On the other hand, when a task is not guaranteed, then the guarantee routine will remove tasks from the system task table of lower criticalness than the new task if those preemptions contribute to the subsequent guarantee of the new task. The lowest criticalness tasks which were preempted, or the original task, if none, are then subject to distributed scheduling. Various algorithms for this combination of deadlines and criticalness, and local and distributed scheduling have been developed and analyzed [2].

Still open are the issues of combining preemptive and non-preemptive tasks and scheduling dependent task groups. We have preliminary ideas for these problems, but they have not been tested.

4 Other Kernel Features

The Spring kernel design has some features similar to current real–time operating systems such as the VRTX kernel [28] or the Hawk kernel [13], but extends these features in many ways including support for multiprocessors, support for a distributed system, a new scheduling algorithm, and careful rework of some features to make them integrate with our new scheduling algorithm. System primitives have capped execution times, and some primitives execute as iterative algorithms where the number of iterations it will currently make depends on state information including available time.

The kernel supports the abstractions of tasks, task groups, dependent task groups, various resource segments such as code, TCBs, TDs, local data, data, ports, virtual disks, non segmented memory, and IPC among tasks. It is possible to share memory (one or more data segments) between tasks. Scheduling is an integral part of the kernel and the abstraction provided is one of a guaranteed task set. The scheduling algorithm handles resource allocation, *avoids* blocking, and guarantees tasks; the scheduling algorithm is the single most distinguishing feature of the kernel. I/O and I/O interrupts are primarily handled by the front end I/O subsystem. It is important to note that the Spring kernel could be considered a back–end hard real–time kernel that deals with deadlines of high level tasks. Because of this, interrupts handled by the Spring kernel itself are well controlled and accounted for in timing constraints. A brief overview of these additional aspects of the Spring kernel is now given.

4.1 Task Management

The Spring kernel contains task management primitives that utilize the notion of preallocation where possible to improve speed and to eliminate unpredictable delays. For example, all tasks with hard real–time requirements are core resident, or are made core resident before they can be invoked with hard deadlines. In addition, a system initialization program loads code, set up TCBs, TDs, local data, data, ports, virtual disks and non segmented memory using the kernel primitives. Multiple instances of a task may be created at initialization time and multiple free TCBs, TDs, ports and virtual disks may also be created at initialization time. Subsequently, dynamic operation of the system only needs to free and allocate these segments rather than creating them. Facilities also exist for dynamically creating new segments of any type, but with proper design such facilities should be used sparingly and not under hard real-time constraints. Using this approach, the system can be fast and predictable, yet still be flexible enough to accomodate change.

The primary task management primitives include:

- CREATE - gets memory for the code segment, loads the code, gets a TD and initializes the TD, and then invokes CREATE-INSTANCE. Again, this command is invoked at system initialization time, or possibly after failures, or when new code must be loaded from disk (a rare occurrence).
- CREATE-INSTANCE - this primitive gets a stack, local data area, TCB (and initializes it), ports, data, and virtual disks for this instance of the task. Multiple instances might be required, in which case the primitive loops to create multiple copies of the required segments, but all the instances share the code and TD.
- DELETE - task and all its associated segments are cleared from memory and the memory space for the code, local data, and data is returned to the non segmented memory segment while all other segments are returned to their respective free lists. Note that all instances of this task are deleted.
- DELETE-INSTANCE - same as delete except code and TD are not deleted.
- SUSPEND - a task is placed on a wait queue; only a task without a hard real-time deadline can SUSPEND. Such non real-time tasks use either idle cycles or cycles dedicated to non real-time tasks, if any. In addition, a SUSPENDed task must release its resources.
- RESUME - this is the mechanism for pulling a SUSPENDed non hard real-time task off the suspend wait list. Any resources it needs have to be reacquired.
- DELAY - a hard real-time task can DELAY itself only if it is willing to be reevaluated for guarantee again and with a new deadline, i.e., a task DELAYING itself is treated as a future arrival of a non-periodic task (it will be reevaluated at that future arrival time or earlier if spare cycles exist).
- ASK - inquires as to the criticalness, the real-time constraint, and/or any other properties of this task. That is, this primitive can read the TD or TCB.
- SET - sets any property of a task or task instance; for example, it can change the real-time constraint description or its value, or it can set the criticalness; as an example, it is possible that the MLC, or a human operator, or the task itself, might be permitted to dynamically modify criticalness.

4.2 Memory Management

Memory management techniques must not introduce erratic delays into the execution time of a task. Since page faults and page replacements in demand paging schemes

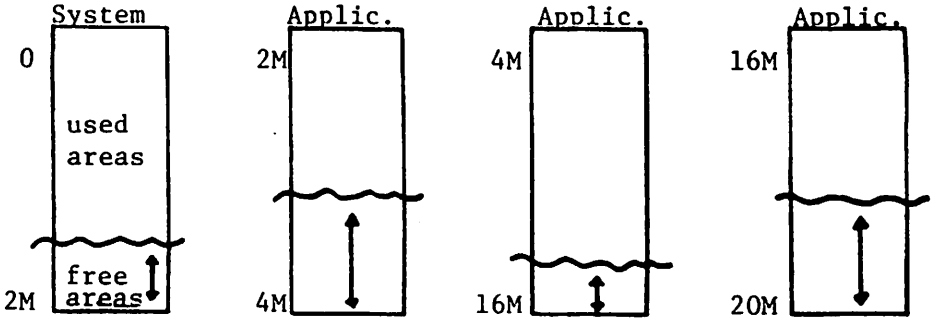
create large and unpredictable delays, these memory management techniques are not suitable to real-time applications with a need to guarantee timing constraints. Instead, the Spring Operating System memory management adheres to a memory segmentation rule with a fixed memory management scheme. Let us now provide an example. We require that there be a reasonable amount of memory at each host[6], and that memory be considered a single virtual address space (see Figure 3). Memory segments include code, local data, data (including shared data), ports, stacks, virtual disks, TCBs, TDs and non segmented memory (See Figure 4). Tasks require a maximum number of memory segments of each type, but at invocation time a task might dynamically require different amounts of segments. The maximum is known a priori. Tasks can communicate using shared memory or ports. However, recall that the scheduling algorithm will automatically handle synchronization over this shared memory or ports. Tasks may be replicated at one or more sites. The Configurator, calling the kernel primitives, initially loads the primary memory of each site with the entire collection of predetermined memory segments. Changes occur dynamically to this core resident set, but it is done under strict timing requirements or in background mode.

To support predictability, a fixed partition memory management scheme is used. The primary memory management primitives are:

- GETMEM - creates a new memory segment of a certain type and size
- RELMEM - releases this memory segment to non segmented memory
- ALLOC - chooses the first free segment of a given type, if none, returns an error
- FREE - segment is returned to its associated free list

Let us now tie together memory management primitives, task primitives, the scheduling algorithm and the fact that we have a multiprocessor. In the Spring kernel, the OS is core resident. The majority of it exists on the system processor with a small piece duplicated on each application processor. A set of M task descriptors and N task control blocks (TCBs) are CREATED at system initialization time and thereafter ALLOCATed dynamically to activated and guaranteed tasks. These segments are on the system processor. A control region for each task that includes code, a user stack, local data, data, ports, virtual disks, is maintained on the application processor. Stack sizes are capped; if a task attempts to push onto a full stack, an error occurs (treated similarly as an attempt to divide by zero).

[6]Many real-time systems are composed of disjoint phases, e.g., in the Space Shuttle [7] there are pre-flight processing, liftoff, space cruising, and descent phases. In this type of non-distributed system, the amount of memory needed is enough to contain the largest phase, not the entire system.



- One virtual address space partitioned among the various memories

Figure 3: Memory Management

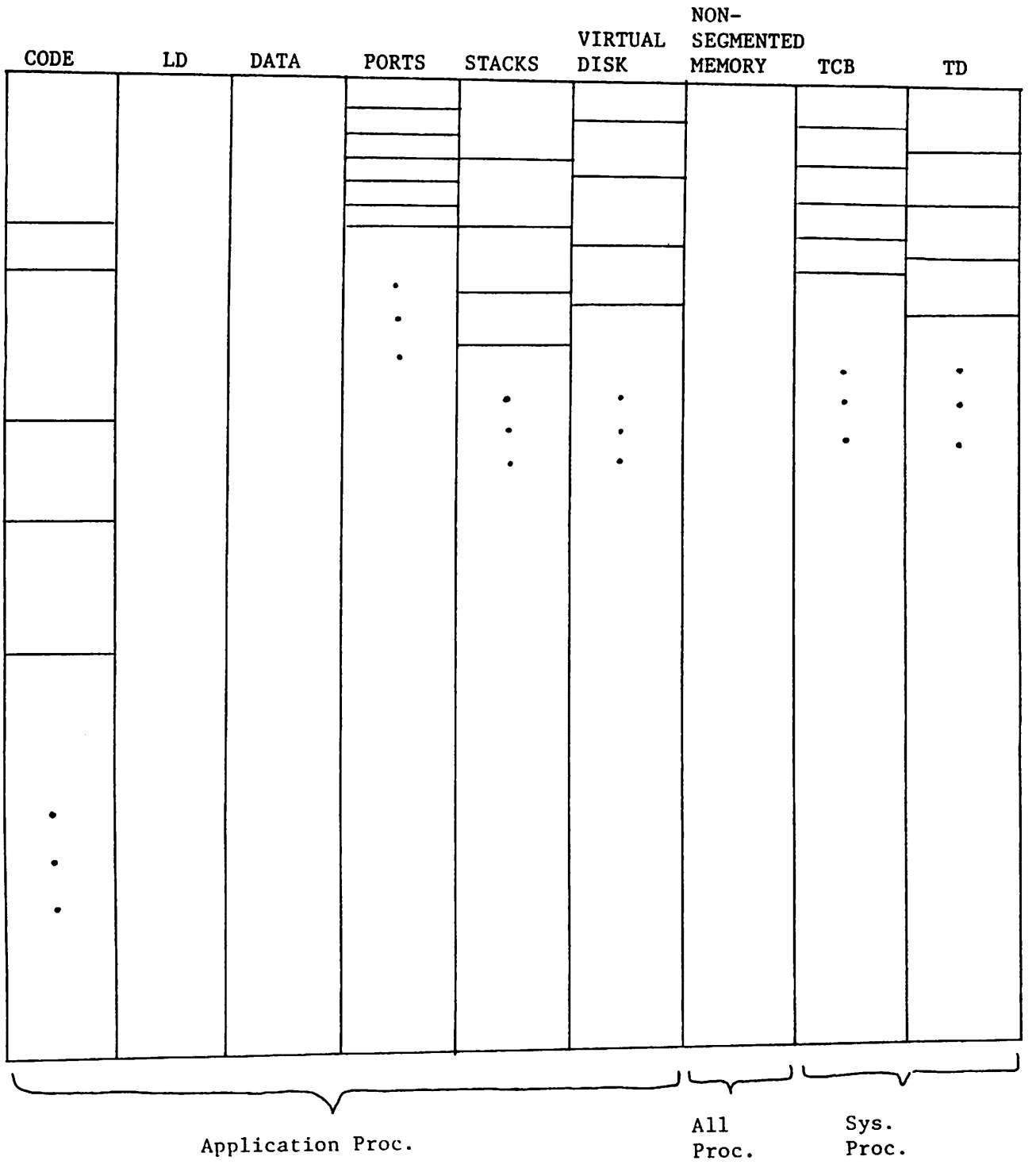


Figure 4: Segments

When a task is activated, any dynamic information about its resource requirements or timing constraints are computed and SET into the TCB; the guarantee routine then determines if it will be able to make its deadline using the algorithm described in section 4. Note that the execution of the guarantee algorithm ensures that the task will obtain the necessary segments such as the ports, data segments, etc. and at the right time. (Again, tasks always identify their maximum resource requirements; this is feasible in a real-time system). If a task is guaranteed it is placed in the system task table (part of memory in the system processor) for use by the application processor dispatcher. A separate dispatcher exists for system tasks which are executing on the system processor. Note that a fixed partition memory management scheme (of multiple sizes) is very effective when the sizes of tasks tend to cluster around certain common values, and this is precisely what our system experiences. Also, pre-allocating as much as possible increases the speed of the OS with a loss in generality. One of the main engineering issues of hard real-time systems is where to make this tradeoff between pre-allocating resources and flexibility. Our approach makes this tradeoff by dedicating front-end processors to both I/O and tasks with short time constraints. As functionality and laxity of tasks increase, we employ on-line, dynamic techniques to acquire flexibility.

Since predictability is of utmost importance, a possible problem is what happens when memory is full. Are predictability and the guarantee violated in this case? The answer is no. The reason is that our scheduling algorithm handles this directly because it takes resource requirements into account when guaranteeing. That is, the task is guaranteed if there is a slot of time in which the task can obtain all its required resources (including memory) and finish executing before its deadline. Hence, the new task attempting to obtain memory is not guaranteed, and other strategies then come into play.

Let's conclude this section with a brief description of the layout of primary memory. The system processor main memory (See Figure 5) contains trap and interrupt vectors and their handlers, two STT collections of pointers for each application processor (this enables the guarantee routine to be working on one set of pointers in parallel with the dispatcher using the second set), the application task TCBs, delay queue pointers, system TCBs, the bidder, the MLC, the Configurator, the local scheduler and guarantee routine, the swapper, the systems processor dispatcher, the page tables (which are used for relocation ease but page faults do not occur since essentially everything is core resident), areas for communication to the I/O and file subsystems, and the rest of the kernel. The application processor's memory (Figure 6) contains the code, local data, shared data, stacks, and ports of application tasks, and a dispatcher which goes to the system memory to ascertain which task to run next, IPC primitives, memory management code, and page tables. The I/O processors memory (Figure 7) contain trap and interrupt vectors and their associated handlers, I/O data structures, device drivers, IPC support to the other processors at

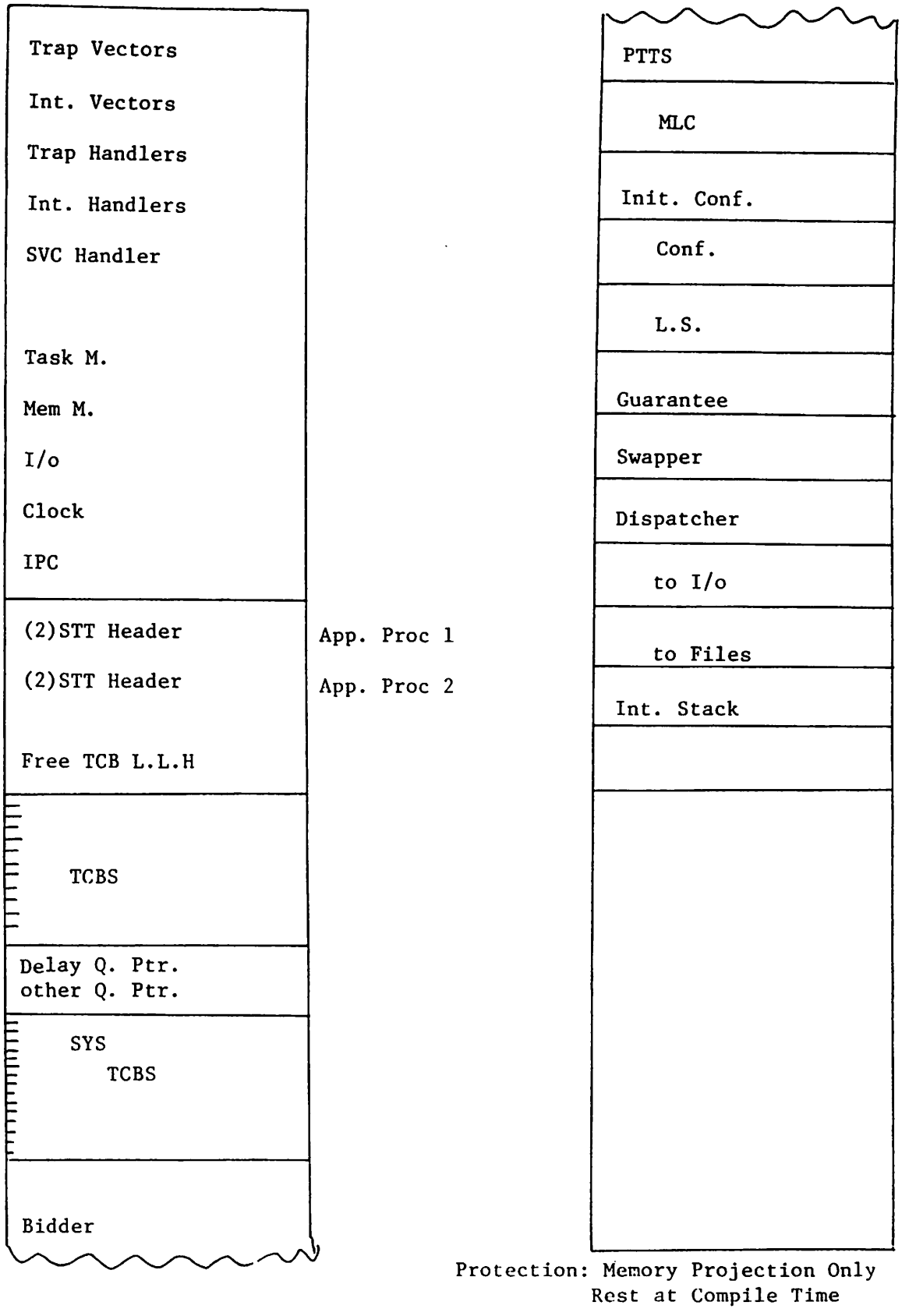


Figure 5: System Processor Memory

this node (signals and moving data in and out of the other memories), and the file system.

4.3 IPC and Ports

The kernel also supports synchronization and communication with 8 interprocess communication (IPC) primitives. The IPC primitives are:

- SEND - send a message to a port and don't wait; the message may have a deadline and this primitive can be seen as implementing a TIMED-DATAGRAM
- RECV - receive a message from a port and don't wait
- SENDW(B) - send a message to a port and wait a time B
- RECVW(B) - receive a message from a port and wait a time B
- ALARM - a most important message; transmit immediately regardless of consequences to all other messages.
- BROADCAST - send to all or to a group of nodes (multicast); this feature is not supported as a time constrained message so it will transmit with lowest importance.
- TIMED-VIRTUAL-CIRCUIT - sets up a one way end-to-end communication channel with guaranteed worst case delivery time

Ports may be local or remote. Receives may only be done from local ports. The kernel to kernel protocol takes care of sending a message to a remote port. The LAN controller implements a time constrained window protocol [34] with special features for alarms, timed virtual circuits and broadcasts, and does DMA into application processor memory to directly extract the message. The higher layers of the protocols are implemented in a bidder module that is running on the system processor. The kernel to kernel protocol is a high performance, minimum copying IPC protocol. There are no automatic acknowledgments on the messages, messages are of fixed maximum size, and responsibility for reliable transmission is left to the application layer. The assignment of a timing constraint to a message transmission is done on the IPC call itself.

Real-time tasks should be programmed with the SEND and RECV primitives whenever possible. In many cases a little thought will enable the programmer to

Objects: Code, data, stack, shared mem. objects, ports

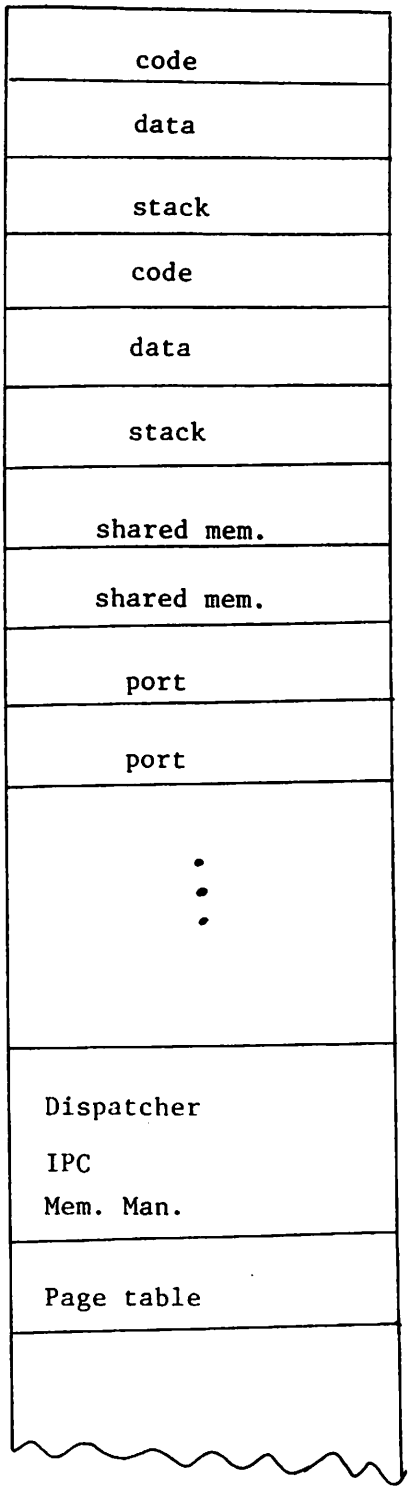


Figure 6: Application Processor

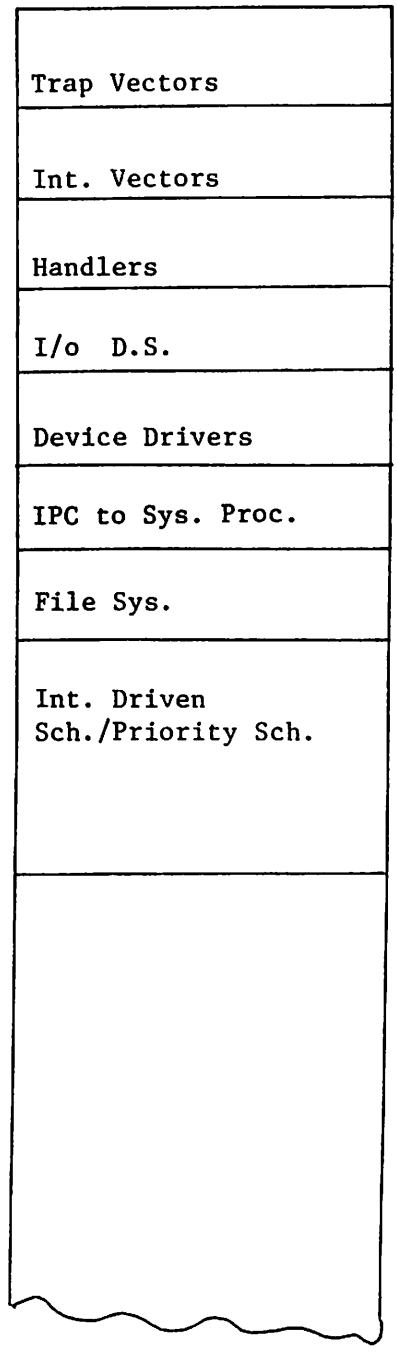


Figure 7: I/O Processor

implement real-time tasks with such primitives. In certain cases it may not be possible or desirable to have a task continue without first waiting for a message. A real-time programmer is permitted to wait for a bounded time B in this case. This bounded time is taken into account in guaranteeing a task. When the bound expires the task must still have some logic to decide what to do in this case. This set of primitives along with the guarantee algorithm prevent a process from waiting past its deadline. That is, scheduling must take an integrated approach and therefore take communication into account.

In timesharing systems semaphores and monitors are typical solutions for guaranteeing mutual exclusion. While it is possible to implement semaphores and their associated P and V operations using the SENDW and RECVW primitives, our full scheduling approach *specifically avoids the need for semaphores by implementing mutual exclusion directly in the schedule.*

Currently, any task that needs to wait for an event or a combination of events must be programmed to do that with the above bounded SENDW and RECVW primitives, or provide enough information to the scheduler so that it is handled by the scheduler. Because of the features of the the Spring kernel including the scheduling algorithm, the preallocation strategy, the dedication of resources, and the programming strategy of not waiting, we expect the system to be largely free of blocking. Synchronization is achieved largely by the scheduler, and if not in this manner, then by bounded waits and polling. The time for the bounded waits is already accounted for in the guarantee process.

For example, if a sensor is producing data, that data is processed and filtered by a dedicated processor in the I/O subsystem. When the data is ready it is passed to a higher level task whose job it is to take action on that data. Hence, a precedence relation exists. Passing the data to a higher level task means that the data is left for the higher level task, which has been guaranteed to run periodically. Now it may happen that when the higher level task runs the data is not actually ready (i.e., the precedence constraint is not met). The higher level task might then do a bounded wait if it were already accounted for in the guarantee, or if the application semantics permits, it just goes on and performs some action such as letting the next periodic instance of the higher level task operate on the data if it becomes available by that time, or performs yet some other action such as using an estimation made from the old data. Again this depends on the application.

Note that the above primitives can also be used to implement a TIMED MONITOR. A timed monitor provides mutual exclusion with a bounded delay and is a useful primitive for real-time applications.

4.4 I/O

Many of the real-time constraints in a system arise due to I/O devices including sensors. The set of I/O devices that exist for a given application will be quite static in most systems. Even if the set of I/O devices changes since they can be partitioned from the main system and changes to them are isolated these changes have minimal impact on the rest of the kernel. Special independent driver processes must be designed to handle the special timing needs of these devices. In Spring we separate slow and fast I/O devices. Slow I/O devices are multiplexed through a front end dedicated I/O processor. System support for this is preallocated and not part of the dynamic on-line guarantee. However, the slow I/O devices might invoke a task which does have a deadline and is subject to the guarantee. Fast I/O devices such as sensors are handled with a dedicated processor, or have dedicated cycles on a given processor or bus. The fast I/O devices are critical since they more closely interact with the real-time application and have tight time constraints. They might invoke subsequent real-time higher level tasks. However, it is precisely because of the tight timing constraints and the relatively static nature of the collection of sensors that we pre-allocate resources for the fast I/O sensors. In summary, our strategy suggests that many tasks which have real-time constraints can be dealt with statically, leaving a smaller number of tasks which typically have higher levels of functionality and higher laxity for the dynamic, on-line guarantee routine.

Note that VLSI techniques (in particular, application specific integrated circuits (ASIC)) have provided the capability for the peripheral subsystem (in all computer systems) to attain a much higher level of intelligence, autonomy and processing power than in the past. For this reason, peripheral functions are becoming more of a factor in overall system design.

4.5 Interrupts

Another important issue is interrupts. Interrupts are an environment consideration which causes problems because they can create unpredictable delays, if treated as a random process, as is done in most timesharing operating systems. Further, in most timesharing systems, the operating system often gives higher priority to interrupt handling routines than that given to application tasks, because interrupt handling routines usually deal with I/O devices that have real-time constraints, whereas most application programs in timesharing systems don't. In the context of a real-time system, this assumption is certainly invalid because the application task delayed by interrupt handling routines could in fact be more urgent. Therefore, interrupts are a form of event driven scheduling, and, in fact, the Spring system can be viewed as having three schedulers: one that schedules interrupts (usually immediately) on the

front end processors in the I/O subsystem (what was discussed above), another that is part of the Spring kernel proper that guarantees and schedules high level application tasks that have hard deadlines, and a third which schedules the OS tasks that execute on the system processor. Interrupts from the front ends I/O subsystem to the Spring kernel are handled by the system processors so this doesn't affect application tasks. In other words, I/O interrupts are treated as instantiating a new task which is subject to the guarantee routine just like any other task. The system processor fields interrupts (when turned on) from the I/O front end subsystem and shields the application tasks, running on the application processors from the interrupts. All OS tasks that run on the system processor have a minimum periodic rate which is guaranteed, but can also be invoked asynchronously due to events such as the arrival of a new task, if that asynchronous invocation would not violate the periodic execution constraint of other system tasks. Asynchronous events are ordered by importance, e.g., a guarantee task is of higher importance than running the meta level controller.

If deadlines for a high level task are very short, if this task must run on an application processor, and if this task is invoked from the I/O subsystem, then time for this task is preallocated. Our scheduling algorithm is flexible enough to handle this variation.

Intraprocessor interrupts like zero divide, overflow, accessing restricted memory, execution of a priviledged instruction, machine failure, and parity errors are considered errors. The current task is in error (maybe through no fault of its own) and may miss its deadline. If an error is recoverable and the executing task can still execute within its worst case time, then our system would still permit it to execute by its deadline and this would not affect other tasks because all of the tasks have been guaranteed with respect to worst case times[7]. If a task needs to execute longer than its anticipated worst case time, say to handle error recovery, then it is possible that it may still execute by using idle cycles up to its deadline, but it is treated as a task without hard real-time constraints. If it does not finish in time, this is considered an error, just like a divide by zero.

5 Miscellaneous Comments about the Spring Kernel

To further clarify some of the points made above, and to address some issues normally discussed when describing kernels, this section contains a miscellaneous collection of comments about the Spring kernel.

[7]In general, real-time system designers and programmers need to prevent long worst case times for tasks.

- One goal of the Spring kernel is to allow various scheduling algorithms to be substituted for each other, depending on the requirements of a particular system. This approach adheres to the *open system* philosophy found in Smalltalk and Cedar. We have designed a standard scheduling algorithm interface so that different scheduling algorithms are easily inserted into different versions of Spring. This will enable the meta level controller (MLC) to easily substitute one algorithm for another; or if a version of Spring does not contain the MLC, then the designer can choose the most appropriate algorithm for his application.

- A strict programming discipline must be followed both for the Spring Kernel and for the applications that use it. For example, extremely large tasks which have a very large variance in computation time requirements must be avoided, and placing abritrary delay statements throughout a hard real- time task is not permitted. Other rules and constraints are being devised as we proceed through the design and implementation of Spring [27].

- In Spring, mutual exclusion is handled essentially "by avoidance" of conflict through the scheduling algorithm. For some tasks, a programmer may be willing to wait for time delta T for a resource. This amount of time could then be accounted for as part of the task's worst case computation time. For the task which executes the wait, there is no problem. However, should it get the resource then, this might affect other tasks which were scheduled for that resource. Hence, we do not permit the delay statement inside application tasks except as the final statement.

- The Spring Kernel also minimizes the need for special signals that inform tasks of start times when those tasks are part of a precedence structure. The reason for this is that the start times are already accounted for in the schedule. This does not preclude the use of signals altogether– signals are used to invoke non-periodic tasks, or new periodic tasks, each of which is subject to the guarantee. A delay statement issued at the conclusion of a task execution is treated as a delayed signal.

- The system design is based on a collection of homogeneous nodes.

- Naming and Protection: The system name space is divided into user space and system space. All system space names are globally unique and do not carry any information about location (transparency). System names are given as follows *node.uniquenumber*. User name space is hierarchical and a name server exists to map user names into system names. A name server at each site also knows where replicas of tasks exist. A cache is maintained at each node that includes information on current locations of tasks. Little run–time protection is provided at this time.

- Global system–wide resource management is accomplished by decentralized facilities. Each facility is composed of up to 4 modules (the dispatchers, local

scheduler, global scheduler and meta level controller (MLC)). Active tasks are located in main memory and are possibly replicated at other sites (in memory or on disk). The global scheduler decides the location of execution for a task if it cannot be executed locally. At times new copies or new tasks might have to be made. MLC is responsible for actually moving tasks to other sites (say for fault tolerance), or for moving tasks into or out of memory. Deadlines are given with respect to the task being in memory. If a task is invoked and it is not in memory, then it is composed of two phases: move into memory and then once in memory it can accept activations that require that it make its timing constraint.

- The kernel also includes real-time clock features such as set-timer, sync-clock, and time-of-day.

6 Summary

The Spring project has a number of major thrusts. One is the development and implementation of the Spring kernel. The first level of design of the kernel is now complete and a plan for rapid prototyping the kernel has been established. As implementation begins we can expect changes, e.g., it may prove better to implement GETMEM as multiple primitives, one for each segment type, rather than having a single primitive for all segment types. As described in this paper, we believe that the Spring kernel contains a number of innovative features that are being investigated as research topics for next generation hard real-time systems.

References

[1] Alger, L. and J. Lala, "Real-Time Operating System For A Nuclear Power Plant Computer," *Proc. 1986 Real-Time Systems Symposium*, Dec. 1986.

[2] Biyabani, S., "The Integration of Criticalness and Deadline Considerations in Hard Real Time Systems," Masters thesis, Univ. of Mass, May 1988.

[3] Blazewicz, J., and Weglarz, J., "Scheduling under Resource Constraints — Achievements and Prospects", *Performance of Computer Systems,* edited by M. Arato, A. Butrimento, and E. Gelenbe, North-Holland Publishing Company, 1979.

[4] Blazewicz, J., "Deadline Scheduling of Tasks with Ready Times and Resource Constraints", *Information Processing Letters*, Vol. 8, No. 2, February 1979.

[5] Blazewicz, J., Lenstra, J. K. and Kan, A. H. G. R., "Scheduling Subject to Resource Constraints: Classification and Complexity", *Discrete Applied Mathematics*, 5, 1983.

[6] Bonuccelli, M. and Bovet, D. P., "Scheduling Unit Time Independent Tasks on Dedicated Resource Systems", *Report S-86-21, Univ. degli studi di Pisa, Instituo di Scienze dell Informazione*, 1976.

[7] Carlow, G., "Architecture of the Space Shuttle Primary Avionics Software System," CACM, Vol. 27, No. 9, Sept. 1984.

[8] Cheng, S., "Dynamic Scheduling in Hard Real–Time Systems," PhD thesis, Dept of Computer Science, UMASS, 1987.

[9] Daniels, D. and H. Wedde, "Real-Time Performance of a Completely Distributed Operating System," *Proc. 1986 Real-Time Systems Symposuim*, Dec. 1986.

[10] Dasarathy, B., "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them," *IEEE Transaction on Software Engineering*, Vol. Se-11, No. 1, January 1985.

[11] Garey, M.R., and Johnson D.S., "Complexity Results for Multiprocessor Scheduling under Resource Constraints", *SIAM J. Comput.*, 4, 1975.

[12] Graham, R.L., Lawler, E.L., Lenstra, J.K., and A.H.G.R. Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey", *Annals of Discrete Mathematics*, 5, 1979.

[13] Holmes, V. P., D. Harris, K. Piorkowski, and G. Davidson, "Hawk: An Operating System Kernel for a Real–Time Embedded Multiprocessor," Sandia National Labs Report, 1987.

[14] Leinbaugh, D., "Guaranteed Response Times in a Hard Real-Time Environment," *IEEE Trans on Soft Eng*, Vol. SE-6, January 1980.

[15] Lenstra, J. K. Rinnooy, A. H. G., and Brucker, P., "Complexity of Machine Scheduling Problems", *Annals of Discrete Mathematics*, 1. 1977.

[16] Liu, C. and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *JACM*, Vol. 20, No. 1, January 1973.

[17] Mok, A., and L. Dertouzos, "Multiprocessor Scheduling in a Hard Real-Time Environment," *Proc. Seventh Texas Conf. Comp Sys*, Nov. 1978.

[18] Ramamritham, K. and J. Stankovic, "Dynamic Task Scheduling in Distributed Hard Real-Time Systems," *IEEE Software*, Vol. 1, No. 3, July 1984.

[19] Ramamritham, K., J. Stankovic, and W. Zhao, "Distributed Scheduling of Hard Real Time Tasks Under Resource Constraints in the Spring System," submitted to *IEEE Transactions on Computers*, Feb. 1986.

[20] Ramamritham, K., J. Stankovic, W. Zhao, "Meta-Level Control in Distributed Real-Time Systems," *Proc.7th Distributed Computing Conference*, Sept. 1987.

[21] Ramamritham, K. and J. Stankovic, "Overview of the Spring Project," COINS Technical report 87-54, 1987.

[22] Ready, J., "VRTX: A Real-Time Operating System for Embedded Microprocessor Applications," *IEEE Micro*, pp. 8-17, Aug. 1986.

[23] Schrage, L., "Solving Resource-Constrained Network Problems by Implicit Enumeration — Non-preemptive Case", *Operations Research*, 10, 1970.

[24] Schwan, K., W. Bo and P. Gopinath, "A High Performance, Object-Based Operating System for Real-Time, Robotics Application," *Proc. 1986 Real-Time Systems Symposium*, Dec. 1986.

[25] Sha, Lui, J. Lehoczky, and R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *Proc 1986 Real-Time Systems Symposium*, Dec. 1986.

[26] Stankovic, J., K. Ramamritham, and S. Cheng, "Evaluation of a Bidding Algorithm for Hard Real-Time Distributed Systems," *IEEE Transactions on Computers*, Vol. C-34, No. 12, Dec. 1985.

[27] Stankovic, J., and L. Sha, "The Principle of Segmentation," Technical Report, 1987.

[28] VRTX/68020 User's Guide, Hunter and Ready, Doc No. 591333001, August 1986.

[29] Ward, Paul, and S. Mellor, *Structured Development for Real-Time Systems*, Yourdan Press, Vol. 1 and Vol. 2, N.Y., N.Y., 1985.

[30] Wirth, N., "Toward a Discipline of Real-Time Programming," *Communications of the ACM*, Vol. 20, No. 8, August 1977.

[31] Zhao, W., Ramamritham, K., and J. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Transactions on Software Engineering*, May 1987.

[32] Zhao, W., Ramamritham, K. and J. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints," *IEEE Transactions on Computers*, August 1987.

[33] Zhao, W. and K. Ramamritham, "Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints," in *Journal of Systems and Software*, 1987.

[34] Zhao, W., J. Stankovic,and K. Ramamritham, "A Window Protocol For Transmission of Time Constrained Messages," *Proc 8th DCS*, June 1988.