REAL-TIME OPERATING SYSTEMS

John A. Stankovic

COINS Technical Report 92-40
May 1992

Real-Time Operating Systems[*†]

Professor John A. Stankovic
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

May 18, 1992

Abstract

Real-time operating systems are an integral part of complex real-time systems. Three general categories of real-time operating systems exist: small, proprietary kernels, real-time extensions to commercial timesharing operating systems, and research kernels. This paper discusses each of these areas focusing on how each of these classes deal with predictability. It is argued that the small, proprietary kernels are predictable, but offer little help to the real-time systems designer and implementor in producing predictable applications. Real-time versions of commercial operating systems like UNIX and Mach offer greater implementation support, but are, *in general*, NOT predictable themselves nor offer enough support to applications which require predictability. This, of course, does not mean that there is *no way* to achieve predictability with these operating systems. It is possible to achieve predictability by very careful design, by using a very limited subset of the overall features provided, and by proving that the features being used for predictability cannot in any way be impacted by any other part of the system. Finally, research kernels are attempting to provide greater design, implementation and evaluation support together with predictability for both the operating system and the application.

[*]This work was funded in part by the Office of Naval Research under contracts N00014-85-K-0389 and N00014-92-J-1048, and NSF under DCR-8500332 and CDA-8922572.

[†]Invited paper for *NATO Advanced Study Institute on Real-Time Computing.*

1 Introduction

Real-time computing systems play a vital role in our society, controlling many types of applications including simple ones such as laboratory experiments and automobile engines, as well as very complex applications such as nuclear power plants, flight control systems, manufacturing processes, and teams of robots working in hazardous environments [1, 19]. In real-time computing the correctness of the system depends not only on the logical result, but also on the time at which the results are produced. Explicitly dealing with time (usually via direct control of equipment that is, in turn, controlling or operating in some larger environment) makes building and analyzing real-time systems difficult. If we can show that a task or set of tasks can meet their timing constraints we say that they are predictable. One important component in producing an effective, predictable real-time system is the *real-time operating system.* The operating system itself must be predictable. This means that the execution times of OS primitives in process management, memory management, IPC, etc. must be small and bounded. In addition, we believe that the real-time operating system should provide a significant amount of direct support for achieving application level predictability, not just operating system predictability. This direct support is missing from current commercial operating systems. There are many commercial and research oriented real-time operating systems. In discussing real-time operating systems it is possible to categorize them into three general groups: small, fast, proprietary kernels, real-time extensions of commercial operating systems, and research oriented operating systems. In this paper we discuss the current state of the art in each of these areas. We point out advantages, problems, and future needs in each of the areas with a particular emphasis on predictability. For tutorial purposes, we also present a more detailed description of one research oriented kernel – the Spring kernel.

2 Small, Fast, Proprietary Kernels

Existing practices for designing, implementing, and validating real–time systems are still rather *ad hoc.* Software engineering practices that advocate modularity and the use of abstract data types are not usually pursued throughout the real–time software production process due to their perceived conflict with real–time requirements. This attitude has permeated the small, proprietary kernels. These kernels are often used for small embedded systems when very fast and highly predictable execution time must be guaranteed [14]. To achieve speed and predictability, these kernels are stripped down and optimized versions of timesharing operating systems. To reduce the run-time overheads incurred by the kernel and to make the system *fast*, the kernel

- has a fast context switch,
- has a small size (with its associated minimal functionality),
- responds to external interrupts quickly (sometimes with a guaranteed maximum latency to post an event but, generally, no guarantee is given as to when processing of the

event will be completed; this later guarantee can *sometimes* be computed if priorities are chosen correctly),

- minimizes intervals during which interrupts are disabled,
- provides fixed or variable sized partitions for memory management (i.e., no virtual memory) as well as the ability to lock code and data in memory, and
- provides special sequential files that can accumulate data at a fast rate.

To deal with timing requirements, the kernel

- maintains a real-time clock,
- provides a priority scheduling mechanism,
- provides for special alarms and timeouts,
- supports real-time queueing disciplines such as earliest deadline and jam a message into the front of a queue, and
- tasks can invoke primitives to delay by a fixed amount of time and to suspend/resume execution.

In general, the kernels also perform multi-tasking and inter-task communication and synchronization via standard, well-known primitives such as mailboxes, events, signals, and semaphores. The kernels often provide very fast interrupt handling and context switches. While all these features are designed to be fast, fast is a relative term and not sufficient when dealing with real–time constraints. Nevertheless, many real-time system designers believe that these features provide a good basis upon which to build real–time systems. This is probably true where the proprietary kernels are most useful, that is, in small embedded applications such as instrumentation, communication front ends, intelligent peripherals and many areas of process control. Then because the application is so simple it is relatively easy to show that all timing constraints are met. Consequently, the kernels provide exactly what is needed[1]. However, as applications become more complex it becomes more and more difficult to *craft a solution* where all timing, computation time, resource, precedence, and value requirements are mapped to a single priority for each task. In these situations demonstrating predictability becomes very difficult. For example, a task may block when it attempts to access a semaphore, new tasks may be dynamically invoked at higher priorities, messages may not be available when a task begins execution, events may be posted very quickly but there may be no guarantee that the processing required to respond to the event will execute in time, etc. Given this large amount of asynchrony, concurrency, and blocking, the unfortunate implementor is required to assign the proper priorities that ensures the system always meets all of its deadlines. Because of these reasons, some researchers believe that current kernel features provide almost no direct support for solving the difficult timing problems, and

[1] Examples of commercials kernels include LynxOS, PDOS, pSOS, VCOS, VRTX32, VxWorks.

would rather see more sophisticated kernels that directly address timing and fault tolerance constraints. Many research kernels are addressing these issues.

Recently there have been efforts to produce seamless real-time kernels that scale from the small, proprietary kernels to large kernels that support the full POSIX/UNIX interfaces (see Section 3). The idea is to let the user select tradeoffs in size, performance and functionality depending on the application. The lowest level of support is being called a nanokernel or alternatively a microkernel. However, it is still not clear if the larger of the seamless kernels will suffer from the same problems discussed in the next section, or if all these problems can be overcome.

3 Real-Time Extensions to Commercial Operating Systems

A second approach to real-time operating systems is the extension of commercial products, e.g., extending UNIX to RT-UNIX [5], or POSIX to RT-POSIX, or MACH to RT-MACH [22]. The real-time version of commercial operating systems are generally slower and less predictable than the proprietary kernels, but have greater functionality and better software development environments. For example, one advantage of RT-UNIX is that it is based on a set of familiar interfaces (standards) that speed development and facilitate portability. However, since many variations of UNIX have evolved, a new standards effort, called POSIX, has defined a common set of user level interfaces for operating systems. In particular, the POSIX P.1003.4, subcommittee is defining standards for real-time operating systems. To date, the effort has focussed on eleven important real-time related functions: timers, priority scheduling, shared memory, real-time files, semaphores, interprocess communication, asynchronous event notification, process memory locking, asynchronous I/O, synchronous I/O, and threads.

Various problems exist when attempting to convert a non real-time operating system to a real-time version. These problems can exist both at the system interface as well as in the implementation. For example, in UNIX interface problems exist in process scheduling due to the **nice** and **setpriority** primitives and its round robin scheduling policy. In addition, the timer facilities are too coarse, memory management (of some versions) contains no method for locking pages into memory, and interprocess communication facilities don't support fast and predictable communication. The implementation problems include intolerable overhead, excessive latency in responding to interrupts, partly but very importantly, due to the non-preemptability of the kernel, and internal queues are FIFO. These and other problems have been solved to result in a real-time operating system that is used for both real-time and non real-time processing. However, because the underlying paradigm of timesharing systems still exists users must be careful not to use certain non real-time features that might insidiously impact the real-time tasks. For example, in [5] they list over 60 RT-UNIX system calls that are not recommended to be used when running a real-time application. This is very disturbing because in converting from UNIX to RT-UNIX the following aspects were changed: scheduling, interrupt handling, IPC, the file system, I/O support, how the user controls resource use, timer facilities, memory management and the basic synchronization assumptions

of the kernel. The juxtaposition of changing almost everything and then ending up with over 60 system calls that should *still* not be used, should lead one to question whether extending a commercial timesharing OS is the correct approach. We believe that it is not the correct approach because too many basic and inappropriate underlying assumptions still exist. This includes optimizing for the average case (rather than worst case), assigning resources on demand, ignoring most if not all semantic information about the application, independent cpu scheduling and resource allocation possibly causing unbounded blocking, etc. On the other hand, the trend to begin with a completely new implementation of UNIX based on microkernels may reduce or eliminate some of the above problems. Consider several more detailed examples from MACH.

MACH is heavily based on lazy evaluation, meaning that you never do anything until it is really needed. One example of this strategy is copy-on-write. Here either a message or part of an address space is not actually copied at the message send time or at address space create time, respectively, but delayed until that message (memory) is actually accessed. On the average this provides excellent performance. The problem is that large amounts of execution time may be required at the *wrong* time to finally perform the copy, causing a task to miss a deadline. Basically, it is unpredictable when slowdowns will occur. Can one eliminate all forms of lazy evaluation to push MACH towards predictability? Yes, but it is difficult because of the overpowering integration of this philosophy in the kernel. Virtual memory is another problem. It is possible to lock pages in memory to remove some of the unpredictability (except, it is nontrivial to decide when to lock and unlock, accounting for the cost of the lock and unlock, and ensuring that the pages are locked in time). Does locking pages, by itself, make the virtual memory part of the system predictable? What about unpredictabilities due to the memory map tables (lookup and maintenance), the MMU TLB entries (present or not), hash table entries used for quick lookup (access time in the table), and indirect problems such as how by locking many pages we might effect the performance of both real-time and non real-time threads needing pages now being drawn from a smaller pool?

Another fundamental problem is that most operating systems want to remove control over resources from the application. These operating systems consider it their prerogative to decide who should get resources for the best average case performance. For example, a multi-level feedback queue will modify the user specified priorities to balance I/O and CPU performance. A real-time application designer who just went through torture to map all the complexities of his application into a set of priorities, and if the system adjusts these priorities, then the analysis and evaluation were for naught. Allowing fixed priorities or another real-time scheduling algorithm helps, but insidious interactions from the non real-time threads, through their resource use and scheduling policy, might slow down the real-time tasks (in some unanticipated way).

Given all these problems for RT-UNIX or RT-MACH can they be used in real-time applications? For real-time applications where missing a deadline has *no* severe consequences, then they can be used. If deadlines must be guaranteed to be met, these operating systems can still be used *if* they are very simple systems and *if* the designers can hand craft a set of priorities that will always work. For example, given 5 independent periodic tasks with certain

periods and deadlines, running only these at fixed priorities on these operating systems can easily be shown to work (it would be just as easy to use the proprietary kernels). However, as we add aperiodics, interrupts from the environment, shared data structures, precedence constraints between tasks, non real-time background processing, etc. assigning priorities such that it will always work becomes a nightmare and the designer is still not certain that lurking problems don't exist due to the underlying timesharing design. Such an approach typically has very high cost and is very difficult to maintain because any change requires a new mapping of priorities. New approaches are needed for real-time computing that challenge the basic assumptions made by timesharing operating systems and provide easy re-analysis upon modifications.

4 Research Operating Systems

While many real-time applications will be constructed with commercial real-time operating systems, as mentioned, significant problems still exist. In particular, the commercial offerings emphasize speed rather than predictability, thereby perpetuating the myth that real-time computing is fast computing [19]. Research in real-time operating systems is emphasizing predictability and argues that totally different approaches are required. In this section we will briefly mention two research projects: the MARS kernel which is based on a time driven model (rather than an event driven model), and the CHAOS system which is based on objects, and conclude with a more detailed description of the Spring kernel which is based on integrated, on-line, planning mode schedulers and global replicated memory to support predictable distributed computing.

The MARS kernel [4, 8] offers support for controlling an application based entirely on time events rather than asynchronous events from the environment. Emphasis is placed on an *a priori* design (including static scheduling) and analysis to demonstrate that timing requirements are met. An important feature of this system is that flow control on the maximum number of events that the system handles is automatic and this fact contributes to the predictability analysis. This system is based on a paradigm, i.e., the time driven model, that is different than what is found in timesharing systems.

The CHAOS system [15] represents an object based approach to real-time kernels. This approach allows easy creation of a family of kernels, each tailored to a specific hardware or application. This is important because real-time applications vary widely in their requirements and it would be beneficial to have one basic paradigm for a wide range of applications. The family of kernels is based on a core that supports a real-time threads package. This core is the machine dependent part. Virtual memory regions, synchronization primitives, classes, objects, and invocations all comprise additional support provided in each kernel. This system does not alter the basic paradigm presented by timesharing systems, but rather tries to work within that paradigm. This example system serves to show that some researchers do not agree that a new paradigm is necessary.

The Spring kernel [18] contains real-time support for multiprocessors and distributed systems. It uses on-line, dynamic planning and retains a significant amount of application

semantics at run time. These features are integrated to provide direct support for achieving both application and system level predictability. A novel aspect of the work is the integration of real-time cpu scheduling and resource allocation. Another is the use of global replicated memory to achieve predictable distributed communication. The abstractions provided by the Kernel include guarantee, reservation, planning, and end-to-end timing support. Spring, like MARS, presents a new paradigm for real-time operating systems, but unlike MARS (to date), it strives for a more flexible combination of off-line and on-line techniques.

4.1 The Spring Kernel

In the remainder of this paper we focus on three main areas of the Spring kernel: process management, memory management and IPC, and stress how they relate to predictability.

4.1.1 Process Management

Processes are the conventional process model seen by the programmer where such things as critical sections, shared memory, and synchronous and asynchronous communication can be used. Collections of processes may be grouped to form a process group with a single timing constraint. This approach enables programmers to use an approach to programming that they are comfortable with and enables the specification of end-to-end timing constraints. However, these programming language abstractions are not convenient for predictable execution because of the unpredictable use of resources and the resulting unbounded and unpredictable blocking that can occur. We use the compiler to transform processes into run time units of execution called tasks which are conducive to predictability. A task is a non-preemptable execution episode of a process with known worst case execution time, resource requirements, value, and timing constraints. A single process is decomposed into a set of tasks (called a task group) with precedence constraints [10]. A process group is transformed into a set of task groups. Once the transformation is complete, each task is completely predictable. The dynamic composition of *predictable tasks* is handled by the on-line planning scheduler [12]. It is important to note that as part of the transformation from processes to tasks, critical sections are eliminated. This is done by identifying the resources protected by the critical sections and associating those resources with a non-preemptable task (again, a piece of a user level defined process). Then, the Spring scheduling algorithm works with these well defined, categorized, and predictable entities called tasks. The scheduler, by planning execution of tasks to avoid resource contention, supports the policy of eliminating critical sections, minimizes context switches, and identifies deadlines that will be missed long before they are actually missed.

Another decision made by the Spring approach is to retain significant application semantics about processes at run time. This enable much more intelligent decisions to be made with regard to actions to be taken if deadlines will be missed. Some of this semantic information is reflected in the information retained for each process.

Processes are characterized by:

- C (a worst case execution time - may be a formula that depends on various input data and/or state information pertaining to a specific process invocation)
- D (Deadline) or period or other real–time constraint
- preemptive or non-preemptive property
- maximum number and type of resources needed (this includes memory, ports, etc.)
- type: critical, essential, or non-essential
- importance level for essential and non-essential processes
- incremental process or not (an incremental task computes an initial answer quickly and then continues to refine the answer for the rest of its requested computation time)
- location of process copies indicating the various nodes in the distributed system and on which processor of each node the process resides,
- precedence graph (describes the required precedence among tasks in a process group)
- fault model that indicates what action to take if this process is not guaranteed to make its deadline.

Scheduling is an integral part of the Kernel and the abstraction provided is one of a *currently* guaranteed task set. It is the single most distinguishing feature of the Kernel. Our scheduling approach separates policy from mechanism and is composed of 4 levels. At the lowest level multiple dispatchers exist; one type of dispatcher running on each of the application processors, and another type executing on the system processor. The *application dispatchers* simply remove the next (ready) task from a system task table (STT) that contains previously guaranteed tasks arranged in the proper order for each application processor. The *dispatcher* on the system processor provides for the periodic execution of systems tasks, and asynchronous invocation when it can determine that allowing these extra invocations will not affect guaranteed tasks, or the minimum guaranteed periodic rate of other system tasks. Asynchronous invocation of system tasks are ordered by importance, e.g., the local scheduler is of higher importance than the meta level controller (see below).

The three higher level scheduling modules are executed on the system processor. The second level is a *local scheduler*. The local scheduler on a node is responsible for dynamically *guaranteeing* that, given the current guaranteed task set, a new task or task group can be scheduled locally so as to meet its deadline. In effect, this algorithm dynamically composes predictable entities (tasks) into a predictable set of entities (the run time execution plan). It does this by ordering the tasks in the STT to reflect the order of their execution from the current time out into the future and in such a way that each task is guaranteed to meet its deadline (taking timing, resource and task value information into account). The logic and implementation details involved in this algorithm are major innovations of our work and details can be found in [12, 13, 16]. In contrast, most other real-time scheduling algorithms are

myopic and only decide the *next* task to execute. Such algorithms may fail catastrophically on overloads because they have no concept of total system state.

The third scheduling level is the *distributed scheduler* which (under certain conditions, e.g., the laxity of the task is large enough) attempts to find a node for executing any task or components of a task group that have to execute on different nodes [11], because they cannot be locally guaranteed. The fourth level is a *Meta Level Controller* (MLC) which has the responsibility of adapting various parameters or switching scheduling algorithms by noticing significant changes in the environment. These capabilities of the MLC support some of the adaptability and flexibility needs of next generation real–time systems. The distributed scheduler and MLC are not yet implemented in the Spring Kernel.

Process management primitives include creating a process which loads the executable image and sets up system data structures such as memory maps, PCB, etc. Currently, this is not done under strict time constraints because we would then require a predictable file system. However, once the process is created it then is eligible for hard real-time scheduling and it is predictable. In the current version, we restrict calling the primitive *create_process* to system initialization time and upon mode changes. Other primitives allow the activation (takes an initialized process and hands it to the scheduler) and deactivation (currently, this process is not to be executed but its image and system structures still exist) of a process. Using the activation and deactivation primitives allows periodic and aperiodic processes to be aynchronously scheduled. Two key aspects of the Spring Kernel are the retention of significant amounts of semantic information and flexibility. Using the *set* primitive, applications can alter any information about a process contained in the PCB. For example, information that can be dynamically changed includes its value, deadline, or fault model. This and other information might vary as a function of system mode and allowing the dynamic updates supports a great degree of flexibility.

4.1.2 Memory Management

Many real-time systems use physical memory techniques in order to facilitate predictability as well as for speed. However, physical memory management has many disadvantages with respect to large, complex real-time systems including difficulty in handling dynamics and protection. The Spring Kernel uses logical memory management to allow for greater protection, dynamic loading, and sharing of portions of address spaces between processes. Logical memory can be implemented in a predictable fashion by using an MMU, but where there are no unexpected page faults (by having the used parts of the address space memory resident).

In Spring, each process has its own logical address space as does the Kernel. A logical address space is supported by using the MMU TLB, partitioned to include the current executing process and the Kernel. In other words, the entire Kernel is always mapped and new process maps are loaded at context switch time. All code and data is memory resident so there is never a page fault. Note that 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 for real–time applications with a need to guarantee timing con-

straints. The Spring Kernel memory management has three main parts. First, allocating the memory resources is done in a careful manner to support predictability. Such resources are either preallocated or handled via the integrated cpu and resource allocation scheduler. More on this below. Second, memory management primitives exist to create memory pools of various types. For example, pools can contain physical pages, various Kernel data structures such as PCBs and ports, logical address spaces, and pools for variable sized blocks. Access to the pools is predictable. Third, there are a number of low level primitives to support the logical address space in a predictable manner. We now discuss each of these parts.

Using Memory Resources Except for the CPU, resources are modeled as memory resources. Many memory resources are created and preallocated at initialization time or mode change. For example, if two tasks of different processes share a data structure in exclusive mode, this data structure becomes a memory resource which is created at initialization time, the same as the tasks. Each of the two tasks is identified as requiring this resource in exclusive mode and the scheduling algorithm uses this resource requirement in its planning. This is a static allocation of resources and a run time guarantee approach. It is also possible to realize the scenario where the resource is not preallocated, but rather allocated at *scheduling* (guarantee time). This is a dynamic allocation and run time guarantee approach. In this case if the allocation cannot be done due to shortage of the resources, then the task is not guaranteed. Finally, it is possible that a task dynamically creates a new resource. However, for such a task the worst case execution time must include the cost of invoking and executing the allocation primitive (which can execute with a bounded cost). If enough resources are not available, then it is an error and the task making the call should perform some appropriate action. Note that the task still completes by its worst case time, thereby not directly affecting the deadlines of other tasks. When a new resource is created it must be made visible to the scheduler and possibly other tasks. This approach is a dynamic allocation without guarantee for *new* resources. In a complicated hard real-time system, application semantics will require all three approaches to be supported. Initially, we are supporting the first approach. We hope to add the other approaches later.

We also point out that tasks are identified *a priori* to require a maximum number of memory resources of different types, but at activation time a task may request fewer resources. This feature potentially allows more tasks to be guaranteed.

Memory Pools Most memory pools are chunks of pre-allocated memory in the Kernel space consisting of a number of fixed-size blocks. Some pools require variable sized blocks, e.g., the graphs describing the process groups and task groups require a size that is dependent on the group size. The purpose of having pools is to support fast, predictable, and dynamic allocation/deallocation of Kernel objects such as PCBs, address translation maps, etc. Predictable primitives for fixed sized pool management include `get_block` and `free_block`. The `get_block` primitive takes a pool's starting address as input and returns a free block's starting address as output. Access time is essentially constant. The `free_block` primitive inserts a block at the head of a pool's free block list. Again, access time is constant. The `init_pool`

primitive structures a pool of a particular type of memory resource. The input parameters to this primitive are the starting address of the pool, total number of blocks contained in this pool, the size of each block in bytes, and the alignment requirement of each block. This primitive then initializes the pool header accordingly, links all the blocks together after aligning them appropriately.

Logical Address Space Support As mentioned above, using logical address spaces has a number of advantages including protection, ease of supporting mode changes, and dynamic loading. The memory management primitives to support logical address spaces include those necessary to initialize memory maps, set page attributes, map and unmap pages, load maps, configure the MMU, make MMU TLB entries valid and invalid, and flush entries out of the TLB. We emphasize that all of these primitives are used to support logical address spaces, *not* virtual memory. Virtual memory with page faults is highly unpredictable and virtual memory where the application locks only certain pages in memory is still not completely predictable. See again the discussion of this fact in Section 3.

To support predictable logical address spaces our solution combines two basic and simple ideas.

- Avoid page faults by preallocating, at process creation time, a physical page for every page in a prgram's address space, and loading that page in memory. This eliminates unpredictability due to page faults.
- Explicitly mange the contents of the TLB to ensure that all memory references experience TLB hits. This eliminates unpredictability due to TLB misses.

When a context switch occurs, the mappings for all the used pages in the logical address space of a process are loaded into the TLB. The TLB always contains the mappings for the portion of the operating system space required to support process execution; they are never flushed. This solution implies that we never make use of the main memory process maps during execution, again contibuting towards predictability. Of course, this approach increases the cost of a context switch. But, the cost is constrained to occur at the context switch time and is completely predictable. It should be noted that while our context switch time might be higher than is commonly expected, other approaches still pay for loading the TLB, however, that cost is not attributed to the context switch time but rather accumulated in a more dynamic fashion as pages are accessed.

Some of the other logical address space primitives are similar to those found in non real-time systems. This includes configuring the MMU which occurs at system initialization time, and initializing maps and setting page attributes which occur at process creation time. Page attributes can also be dynamically changed.

4.1.3 Interprocess Communication

In conventional systems, IPC can be unpredictable due to the potentially unlimited blocking time of applications synchronizing or waiting for messages. Complemented by our process

to task mapping techniques and the scheduling approach, our IPC subsystem is distinct in that it provides predictable and bounded synchronous communication in a hard real-time environment. The Spring IPC mechanism provides a relatively conventional interface with message-passing using ports, but a significantly unconventional implementation that supports predictable real-time communication. Ports are kernel-protected memory objects (and their associated control information) that hold units of data called *messages*. Processes can communicate with each other by placing messages into ports and removing messages from ports. Ports have bounded capacity for predictability, and programmer-specifiable overflow and queueing policies. Messages have fixed sizes, and strict copy-by-value semantics. Messages can have *deadlines* that determine when they must be delivered to a port.

Ports and messages are *typed* in two ways, both based on the kind of communication employed. The components of the type are:

1. *Task Type.* This mirrors the types available to tasks: critical, essential, soft real-time, or non real-time. This is in order to prevent non-guaranteed tasks from affecting guaranteed tasks' timing properties.

2. *Semantic Type.* This describes how messages in a port are to be used. These types are asynchronous, synchronous, request, and reply.

In distributed communication, a *connection* must be established between the sender and receiver, in order for two processes to communicate. The connection is the network bandwidth required to *send* to the port. Thus, it is the sending side that must provide the connection. We have identified two types of communication abstractions [2] to cater to the needs of the system.

Real-time virtual circuits (RTVC's) are dedicated channels that guarantee network access within a bounded amount of time. RTVC's are generally used by hard real-time tasks, and are allocated appropriately. Critical tasks have their RTVC's pre-allocated at system boot time; essential tasks have their RTVC's allocated at guarantee time.

Real-time datagrams (RTDG's) are communication channels that provide network access on a *best efforts* basis. RTDG's attempt to deliver a message within its deadlines, but no guarantee is made. RTDG's are used by soft and non real-time tasks, and are allocated at run time.

The primitives we provide to the programmer are the following:

- *Asynchronous Send and Receive.* The sender does not wait for the receiver, and also does not wait to see if the message can be queued at the destination port; a receiver does not wait if no messages are available. Messages sent have a *deadline* that they *should* be delivered by.

- *Synchronous Send and Receive.* The source process suspends after a send until the destination process performs a receive on the port and dequeues the message. The message sent has a *deadline* that it must be delivered by.

The sender of a message specifies a deadline – this is the time by which the IPC system should deliver the message. The delivery is guaranteed to arrive on time for synchronous communication involving hard real-time tasks. For asynchronous communication the system does its best to deliver the *message* on time. However, note that the deadlines of the individual hard real-time tasks are still guaranteed.

We also provide *request-response* primitives, but do not describe them here due to space limitations. It should suffice to say that request-response is a higher-level abstraction that can typically be constructed using synchronous send and receive.

Mapping Processes to Tasks The issue of mapping processes to tasks arises in inter-process communication. Synchronous communication conventionally implies that a task can be blocked for an indefinite period. For hard real-time tasks, unpredictable blocking can obviously not be allowed. To overcome this, we take advantage of the semantics of communication and the Spring system paradigm, mapping *processes* to *tasks*. To review, processes are the conventional process model abstraction seen by the programmer. Tasks are the units of execution that are scheduled and run by the Spring system. Synchronous communication in hard real-time tasks require recognition by the compiler and support in the scheduler.

A group of programs that cooperate and communicate among themselves form a *process group*. Each process is be decomposed into tasks that have precedence constraints attached to them. The precedence constraints allow the scheduler to construct a schedule that, if feasible, will guarantee that the entire process group will complete on time. In the distributed case, before any task of a process group is run, the local scheduler knows the execution plan of each task in the entire process group and their allocated communication channels.

Thus, at run-time, if a guaranteed task performs a synchronous receive, the scheduler knows *when* to schedule that task such that the message is *guaranteed* to be present. Thus, suspending a process defines a boundary between two tasks. The duration of the suspension time is determined by the scheduler when the process group as a whole is guaranteed.

In addition to compiling code, the Spring Compiler Environment, called The *Software Generation System* (SGS), identifies the correspondence between matching synchronous primitives within a process group. Information is passed between the SGS and the scheduler so that the scheduler can identify the precedence constraints between tasks.

Restrictions on Synchronous Communication Since synchronous communication affects scheduling decisions made for guaranteed tasks, we must maintain some restrictions on use of the synchronous IPC primitives. For example, we cannot allow a non-guaranteed task to communicate synchronously with a guaranteed task, as it would make the guaranteed task unpredictable. We cannot even allow a critical hard real-time task to synchronously communicate with an essential task. Because essential tasks are guaranteed dynamically, there is no *a priori* guarantee that any essential task will *ever* execute.

Thus, we must restrict synchronous communication in hard real-time tasks such that only tasks of the same type can synchronously communicate. In other words, critical tasks can only use synchronous communication with other critical tasks. Essential tasks may

communicate synchronously only with other essential tasks. We do not need to restrict synchronous communication between soft and non real-time tasks, as they do not affect scheduling decisions.

We have implemented the IPC primitives using Systran's Scramnet distributed globally shared memory architecture [21] as a platform for our distributed system. Scramnet is a replicated global shared memory architecture that uses a fiber optic register insertion ring, running at 150 Mbits/sec. Each node has 2 MB of shared common memory, and writes to this memory are broadcast (circulated) about the ring by hardware. The main advantages of using a global shared memory architecture are that implementation of IPC primitives is easy (there are no levels of protocol stacks) and the resultant IPC is predictable. We are currently trying to exploit the global replicated memory for distributed scheduling and fault tolerance. It is also possible to scale the architecture for high performance computing by creating a 2-dimensional grid of rings where each node is connected to two rings [20]. We will not discuss this option here since our current configuration has only one fiber optic register insertion ring with 2 Mbytes of replicated memory.

5 Summary

Most critical, real–time computing systems require that many competing requirements be met including hard and soft real–time constraints, fault tolerance, protection, and security requirements [19]. In this list of requirements, the real–time requirements have received the least formal attention. We believe that it is necessary to raise the real–time requirements to a central, focusing issue. This includes the need to formally state the metrics and timing requirements (which are usually dynamic and depend on many factors including the state of the system), and to subsequently be able to show that the system indeed meets the timing requirements. Achieving this goal is non-trivial and will require research breakthroughs in many aspects of system design and implementation. For example, good design rules and constraints must be used to guide real–time system developers so that subsequent implementation and *analysis* can be facilitated. Programming language features must be tailored to these rules and constraints, must limit its features to enhance predictability, and must provide the ability to specify timing, fault tolerance and other information for subsequent use at run time. Execution time of each primitive of the Kernel must be bounded, and the operating system should provide explicit support for meeting application level requirements including the real–time requirements. More work is required in many areas of real-time operating systems including scheduling, I/O, predictable IPC, and robustness. The hardware must also adhere to the rules and constraints and be simple enough so that predictable timing information can be obtained, e.g., caching, memory refresh and wait states, pipelining, and some complex instructions all contribute to timing analysis difficulties. An insidious aspect of critical real–time systems, especially with respect to the real–time requirements, is that the weakest link in the entire system can undermine careful design and analysis at other levels. Research is required to address all of these issues in an integrated fashion.

6 Appendix – A Partial List of Research Questions

Although there are many research projects on real-time operating systems many open questions remain. A partial list of such questions is as follows:

- how deterministic can or should the OS be
- how can the OS support predictable distributed communication
- what support should be provided for end-to-end timing constraints, fault tolerance, safety, security, etc.
- how fault tolerant should the kernel itself be
- can a timesharing interface to an OS be made suitable for hard real-time with proper implementation
- should a real-time OS be seamless (from a micro-kernel all the way up to a large OS)
- what support is needed for monitoring, clock synchronization, etc.
- is a local memory or shared memory model more suitable for multiprocessor architectures
- how can we exploit distributed shared memory and/or replicated global memory (sometimes called reflected memory)
- what are the correct interfaces to robotics, RTAI, Vision, high speed networking, multimedia, etc.
- are standards appropriate for real-time OSs at this time
- where is an OS (and architecture) impacted by the need to design for worst case rather than average case
- where is an OS (and architecture) impacted by the need to design for the most important case rather than the most frequent [7]
- what functionality should be in the OS level and what in the application level
- is a microkernel a good idea and if so what mechanisms and functionality should be in the microkernel
- what abstractions should be supported by a real-time kernel (e.g., real-time POSIX is considered unsuitable by many researchers, but what is missing? Do the correct abstractions include deadlines explicitly, guarantee, reservation, and fault tolerance?)
- where is the dividing line between policy and mechanism and should the notion of separating policy and mechanism be applied to all functions in the kernel

- what should be the unit(s) of execution (independent tasks versus groups of tasks all at various granularities)
- can object oriented real-time OSs be effective; are they the next generation operating system

7 Acknowledgments

Many students have worked on various parts of the Spring Project including the Kernel. I wish to thank K. Arvind, S. Biyabani, V. Cheng, M. Decao, E. Gene, M. Kuan, L. Molesky, E. Nahum, D. Niehaus, C. Shen, P. Shiah, F. Wang, W. Zhao, and G. Zlokapa. I would especially like to thank Prof. Ramamritham for co-directing the project with me for many years.

References

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

[2] Arvind, K., K. Ramamritham, and J. Stankovic, "A Local Area Network Architecture for Communication in Distributed Real-Time Systems," *Real-Time Systems*, Vol. 3, No. 2, May 1991.

[3] Biyabani, S., J. Stankovic, and K. Ramamritham, "The Integration of Criticalness and Deadline In Scheduling Hard Real–Time Tasks," *Real–Time Systems Symposium*, December 1988

[4] Damm, A., J. Reisinger, W. Schnakel, and H. Kopetz, "The Real-Time Operating System of Mars," *Operating Systems Review*, pp. 141-157, July 1989.

[5] Furht, B., D. Grostick, D. Gluch, G. Rabbat, J. Parker, and M. Roberts, *Real-Time Unix Systems*, Kluwer Academic Publishers, Norwell, Massachusetts, 1991.

[6] 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.

[7] Jensen, D., "The Kernel Computational Model of the Alpha Real-Time Distributed Operating System," in *Mission Critical Operating Systems*, edited by A. Agrawala, K. Gordon and P. Hwang, IOS Press, 1992.

[8] Kopetz, H., A. Demm, C. Koza, and M. Mulozzani, "Distributed Fault Tolerant Real-Time Systems," *IEEE Micro*, pp. 25-40, February 1989.

[9] Molesky, L., C. Shen, and G. Zlokapa, "Predictable Synchronization Mechanisms for Real-Time Systems," *Real-Time Systems*, Vol. 2, No. 3, pp. 163-180, September 1990.

[10] Niehaus, D., "Program Representation and Translation for Predictable Real-Time Systems", *Proc. Real-Time Systems Symposium*, Dec. 1991.

[11] Ramamritham, K., J. Stankovic, and W. Zhao, "Distributed Scheduling of Tasks With Deadlines and Resource Requirements," *IEEE Transactions on Computers*, Vol. 38, No. 8, pp. 1110-1123, August 1989.

[12] Ramamritham, K., J. Stankovic, and P. Shiah, "Efficient Scheduling Algorithms for Real–Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, pp. 184-194, April 1990.

[13] Ramamritham, K. and J. Stankovic, "Scheduling Results in the Spring Project," Chapter in *Foundations in Real-Time Computing: Scheduling and Resource Management*, editor Andre van Tilborg, Kluwer Academic Publishers, 1991.

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

[15] Schwan, K., A. Geith, and H. Zhou, "From $Chaos^{base}$ to $Chaos^{arc}$: A Family of Real-Time Kernels," *Proc. 1990 Real–Time Systems Symposium*, pp. 82-91, December 1990.

[16] Shen, C., K. Ramamritham, and J. Stankovic, Resource Reclaiming in Real-Time, *Proc Real-Time System Symposium*, pp. 41-50, December 1990.

[17] Stankovic, J., and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real–Time Operating Systems," *ACM Operating Systems Review*, Vol. 23, No. 3, pp. 54-71, July, 1989.

[18] Stankovic, J. and K. Ramamritham, "The Spring Kernel: A New Paradigm for Hard Real-Time Operating Systems," *IEEE Software*, pp. 62-72, May 1991.

[19] Stankovic, J., "Misconceptions About Real–Time Computing," *IEEE Computer*, Vol. 21, No. 10, October 1988.

[20] Stankovic, J., "SpringNet: A Scalable Architecture For High Performance, Predictable, Distributed, Real-Time Computing," Univ. of Massachusetts, Technical Report, 91-74, October 1991.

[21] SYSTRAN Corporation, "Scramnet Network Reference Manual," Dayton, Ohio, 45432.

[22] Tokuda, H., and C. Mercer, "ARTS: A Distributed Real–Time Kernel," *ACM Operating Systems Review*, Vol. 23, No. 3, July, 1989.