EDITORIAL: REAL-TIME KERNEL INTERFACES

BY
PROF. JOHN A. STANKOVIC
Univ. of Massachusetts

Current real-time kernels present inadequate, incomplete, and missing functionality in their interfaces. As a result of these poor interfaces, real-time systems are difficult to design, maintain, analyze, and understand. Here, I present examples of the poor interfaces and suggest improvements for scheduling, memory management, and interprocess communication. Basically, I am suggesting a need for a much higher level of functionality than is currently provided. This argues against the trend for microkernels for real-time operating systems (at least in one respect as discussed below).

Scheduling: Most real-time kernels provide a fixed priority scheduling mechanism together with a few system calls that can adjust priorities. This works well when tasks' priorities are fixed. However, in general, this mechanism is *inadequate* because many systems require dynamic priorities and mapping a dynamic scheme onto a fixed priority mechanism can be very inefficient. Let me give an example. Suppose we are using an off-the-shelf, black box, real-time POSIX compatible kernel and we are given a fixed priority scheduler with 16 levels. The application programmer might want to implement earliest deadline scheduling for aperiodic tasks. When the first task arrives, the policy module (an application task) must assign a priority based on the relationship of deadlines of all the tasks. In this case it is the only task so the policy module could assign any of the 16 numbers. Suppose it assigns the highest priority. Then a second task arrives with an earlier deadline. No problem, the policy module simply lowers the priority of the first task and gives the second task the highest priority. However, because the priority mechanism is in the interface of a black box kernel, the policy module must first issue a system call to remove the previous task, and then two more system calls to reinsert the two tasks. Further, the policy module is required to keep track of all the tasks and their deadlines, *outside the kernel*, so that it can re-adjust the priorities when certain arrival patterns make it necessary. In the worst case a new arrival might require every task to be removed and reinserted. Of course, it is possible to significantly improve the performance just described by various techniques such as adding system calls that insert and or remove a set of tasks, or a call that simply shifts a task's priority. Even with these additional interface primitives, the use of a fixed priority scheme can be inefficient because it is the wrong model for dynamic priority scheduling.

I propose that the underlying support should be a dispatcher that simply removes the next task from an ordered list, checking a start time field for the task to make sure that the next task to execute has an arrival time that has passed. This latter feature is useful for periodic tasks and aperiodics with future start times. The ordered list itself is maintained by the policy module, but the dispatcher has direct access to it, obviating the need for separate copies of information. The policy module can be whatever the application requires but for efficiency becomes part of the kernel in the sense that its data structure is visible to the dispatch mechanism. With this scheme we also eliminate the granularity issue where there are not enough priority levels, e.g., only 16 in the above example. In that case, multiple tasks are usually mapped to a single level, thereby possibly violating the scheduling policy such as earliest deadline. With the suggested approach, the granularity issue is now a function of the size of the time field which, in my opinion should always be explicitly represented in a real-time system anyway (and not converted to a priority), and needs to be at least 32 bits. Various real-time policy modules should be available at load time, or even at run time. FCFS and round robin should not be provided. These are not real-time scheduling algorithms and if users really want them, let them provide them. To emphasize the real-time nature, the policies available should include earliest deadline, least laxity, rate monotonic, sporadic server, stack resource policy, and the Spring scheduling algorithm. This is exactly the opposite of the approach taken by real-time POSIX.

Fixed priority scheduling is also *incomplete* because it deals only with the CPU resource. Since we are interested in when a task completes we must consider all the resources that a task requires. An integrated view of resource management should be part of the interface including the ability to specifically identify the needed resources and to reserve them for the (future) time when they will be needed. I suggest that the interface support the specific *reservation* of sets of resources in an atomic fashion for a given duration. For example, I might specify that process A requires 10 milliseconds of CPU and data structure resources B and C, and 3 new pages of memory. The system should be able to perform the *atomic* reservation of this entire set of resources for a duration of 10 milliseconds within a time window that extends from the current time to the deadline. It should also be possible to specify a start time for the reservation such as please reserve the execution beginning at 3 o'clock. The reservation should proceed via a best effort approach and if it is not possible, then a specified policy (part of the interface) should be invoked. Reservations need not be the only resource management strategy employed in the system. For example, a priority ceiling protocol manages resources by imposing a structure on how resources can be acquired without reserving them atomically. Both reservations and means for identifying ceilings on resources could specifically be in the interface. With this level of support an application designer can develop a system more quickly, analyze it easier, and understand its operation better.

Fixed priority scheduling has *missing functionality* because it substitutes a single priority number for possibly a set of issues such as the semantic value of completing the task, the timing constraint of the task, and fault properties of the task. Further, a fixed priority ignores the fact that semantic information is often dynamic, i.e., a function of the state of the system. I propose that as part of the interface deadline or other timing constraint, value, fault semantics, and precedence constraint information be specifically input to the operating system (when necessary), and be easily alterable by application level policy modules. In this way, the system could implement sophisticated decision making modules and avoid anomalies that occur when you map multiple parameters onto a single number, or erroneously assume that the relative priorities of tasks are fixed. All real-time applications require support for timing and reliability. Why not explicitly provide it, rather than leaving it to the designers?

Memory Management: Some real-time kernels support only a physical address space. This is a bad idea because of poor protection between modules and because it has poor maintainability properties for long lived real-time systems. Other kernels support virtual memory. This is also a bad idea, *even* if you are allowed to lock pages into memory. In these systems even if certain pages are locked, non-determinism arises from TLB misses, map table maintenance, possible blocking due to paging I/O from other pages, and secondary effects from less available pages. The interface should support multiple logical address spaces for protection and maintenance (new programs can easily be loaded dynamically) reasons. However, the entire address space should be loaded so that there is never a page fault, never a TLB miss, etc. The system should also provide an interface for manipulating the address space of a process. This would support controlled overlap of address spaces, thus supporting shared data, as well as making hardware control and status registers directly available to processes requiring them.

IPC: Many real-time kernels support standard synchronous or asynchronous IPC or both, and send information across node boundaries via layers of communication protocols implemented in a non real-time fashion. If queuing occurs, it is FCFS. There is nothing real-time about these primitives at all. We need to support real-time virtual circuits where delivery is guaranteed by a deadline and where the deadline, value, and reliability requirements of the message are part of the interface. Then an application task requiring end-to-end delivery by a deadline must request a real-time virtual circuit. The implementation is likely to require a reservation of a set of resources. If an application task can live with a best effort delivery of the message, then it should invoke a real-time datagram with a similar interface as for the virtual circuit except that the semantics of delivery is best effort rather than guaranteed. It is important to note that these communication primitives by themselves are not sufficient since the processes making the calls must also be scheduled.

Many other aspects of what a real-time kernel interface should be are not addressed in this short note, but they all need to put reservations, timing constraints, value, and reliability requirements explicitly into the interface. Further, I should note that many implementation issues such as minimizing interrupt latency, interrupt service time, preemption latency, and context switch time are also very important real-time kernel issues. However, these issues are implementation issues and are not part of the interface. Further, the execution time costs required for these factors are relative to the application at hand.

Finally, let's consider a current hot topic, microkernels for real-time. Basically, the use of microkernels is a good implementation technique, so I do not argue against using microkernels. However, I think that the current designs are attempting to find the core ingredients from the *wrong*, current, higher level interfaces. An interface like I suggest here should be the basis upon which to develop a microkernel should you choose to build your system that way. I am well aware that many people advocate very simple real-time kernels or even microkernels where most of the functionality I suggest would clearly be outside the kernel. I think that this somewhat irrelevant for real-time systems designers. These designers require certain system run-time support, and it doesn't really matter to them whether its a black box microkernel with all the functions they need running on top of it, or an open system model that lets them compose what they require into a run time system. Remember, it is the entire embedded system which is important, not the kernel nor microkernel, per se. Another problem with all the recent emphasis on real-time microkernels is that such simple kernels focus real-time operating system efforts in the wrong place. We already know how to build and provide interrupt handlers, priority schedulers, semaphore support, standard IPC mechanisms, etc. These efforts are simply implementing functionality that has existed for many years. They are re-inventing the wheel. However, with these simple functions as primitives, real-time systems design and implementation has not progressed satisfactorily. It is still to costly and error prone to build real-time systems. The real effort should be in proving applications with explicit algorithmic support for meeting deadlines, meeting fault tolerance requirements, and meeting monitoring, analysis, and understandability requirements. With the correct primitives, we can elevate the designer from having to carefully select priorities so that no matter what happens in the system (including all forms of blocking, interrupts, new task arrivals, failures, etc.) the selected priorities will be such that all deadlines are met. Rather, there should be algorithmic run-time support that attains the same goal. Further, choosing a non real-time interface for real-time systems (which is what these low level, priority based interfaces are) provides implementation problems requiring users to *get around* the problem.

EDITORIAL: RESOURCE ALLOCATION IN REAL-TIME SYSTEMS

by
Prof. John A. Stankovic

Large real-time systems are complex. In these systems there are usually many periodic and aperiodic tasks, a high degree of concurrency, and significant asynchronous behavior. Requirements often demand careful design and analysis so that the system is predictable. Unfortunately, much of the analysis has focussed on a too simplistic model. For example, the basic rate monotonic result requires many simplistic assumptions which are not met in practice. Consequently, for a complex system it does not matter that you have a beautiful theorem with precise analysis because the coverage of the assumptions compared to the real system is low. While the basic approach may still be useful for an approximate analysis, it must be noted that any approximate analysis is more limited for a real-time system than for a general purpose timesharing system because designers require precise analysis with respect to deadlines of specific tasks. Another example of simplistic analysis is earliest deadline scheduling. Simple formulas can again provide precise results based on assumptions such as all tasks are preemptable at any point, there is no sharing of resources, and no overloads will ever exist. Again, if the manner in which these assumptions match the real system is poor than the precise analysis is not precise at all for the system in question! One key problem with the basic rate monotonic and earliest deadline policies and their analyses is that they *fail to recognize that all resources must be addressed by the analysis and not just the CPU.* In other words, what is required is an integrated resource allocation model for real-time where

- all resources are addressed,
- the assumptions made in the model and the actual system (including the environment) match to an acceptable degree,
- the run time system supports the model in a precise manner,
- it is analyzable, and
- the system is understandable.

In this short note, I briefly consider three alternatives for an integrated resource allocation approach. One, is the rate monotonic scheduling with priority ceiling based on an underlying on-demand resource allocation model [9]; two, is the planning approach (typical of the Spring [10] and Maruti systems [2]) which is not based on an on-demand model; and, three, is static real-time systems based on table driven, cyclic [4, 5] or time driven [3] approaches. To a large extent these are also not based on the on-demand model.

Rate Monotonic with Priority Ceiling:

Many useful extensions to the basic rate monotonic algorithm have been produced including the priority ceiling protocol which effectively creates an integrated resource allocation model. It can be said that this protocol works in conjunction with an *on-demand* paradigm found in many systems including all general purpose timesharing systems.

The on-demand approach has come into existence mainly for efficiency. It is a lazy allocation approach, i.e., resources are assigned only when actually required so that the system never wastes an assignment that will not be used. The penalty paid for this is non-deterministic blocking. For time sharing environments this blocking is usually quite acceptable, on the average. But in a real-time environment, the non-deterministic blocking is the source of analysis difficulty. Note that *all* the other real-time approaches mentioned in this editorial carefully *avoid* these blocking complexities. Since blocking is permitted in the on-demand model, we must impose some structure on this blocking, e.g., as priority ceiling does, to bound the blocking time. Doing this enables us to re-introduce nice analysis properties, but does so with severe limitations in *some* situations.

Advantages of the priority ceiling protocol running on an on-demand uni-processor system include:

- all resources are addressed
- run time system supports the model in a precise manner
- it is analyzable
- it is understandable at a high level meaning that given a set of tasks we know if they will be guaranteed to make their deadlines
- often allows very high utilization on a uni-processor and possibly in a multiprocessor, e.g., when very few, short global critical sections exist
- robust in the sense that the run time mechanism is dynamic where, for example, worst case execution times for several tasks might have been computed erroneously yet deadlines are still met because other tasks dynamically execute at less than their worst case times
- overload is not on a specific task overrunning a specific boundary as in static systems, but is expressed as an overall utilization in excess of a utilization bound
- best technique available for on-demand system model

It is important to note that many of the disadvantages listed below do not always exist, but when the conditions mentioned do exist, then the limitations can be severe. Disadvantages mainly arise from (1) pessimistic assumptions made by the analysis, and (2) as systems scale blocking becomes complex[1]. The disadvantages include:

- as systems scale in size the assumptions made in the model often don't match the actual environment
- it is often difficult to understand the actual execution histories of the system which are completely dynamic, giving rise to difficulty in testing, reproducibility, and for replica determinism. The understandability issue is extremely important in safety critical systems and is one of the main reasons static designs are still so widely used.

[1]Baker has proposed a modification where a degree of non-blocking is added to the basic priority ceiling protocol.

- if the highest frequency task (or the top n tasks) is very fast and has non-negligible blocking factors, then those blocking times are amplified (see the formula used in the analysis) or if there are large critical sections anywhere in any program then utilization can be very low. An implication of these issues is that for most sophisticated real-time systems the answer to most analysis might be that the system is not schedulable. In that case other techniques will be necessary anyway, e.g., by using more semantic information about the application, or significantly overdesigning with very low utilizations.

- addressing the critical sections used by the kernel and understanding how any inhibited preemptions or interrupts in the kernel impact the analysis has not adequately been addressed, e.g., the tasks may make system calls which subsequently turns off interrupts when in a critical section delaying the arrival of work which may invalidate one of the assumptions of the analysis

- if resources are multi-unit then more complicated schemes are necessary with the result of increasing run time cost and being very pessimistic about when a task can execute

- not easily extensible to multiprocessing [6]; this is a critical limitation

- not easily extensible to distributed systems

- not easily extensible to precedence constraints

- employs an artificial method for integrating the value of a task with its frequency, and given a particular assignment might cause an important task from missing a deadline in overload; One technique often suggested is artificially reducing the period of an important task and breaking up its execution time, but this is not a good solution when a task has critical sections. It is difficult to ensure that the critical section is not held across task executions.

- not easily used for systems which must prevent jitter

- may have high numbers of context switches

- integration of periodics and aperiodics forces everything to look like a periodic task which can be very inefficient for tasks which are rarely invoked

Planning: The planning approach assumes a different underlying system model than the on-demand model. It is based on reserving resources into the future [7], thereby avoiding blocking. The planning approach also carefully isolates interrupts thereby ameliorating another major analysis and understandability problem. These two main features are very similar to the main features found in static real-time systems. The main differences between static systems and planning systems is that planning based systems are more dynamic and flexible, but less predictable, while the static systems are less flexible, sometimes too costly in resources, but are more predictable.

Advantages of the planning approach accrue due to its very general nature and because it takes a system-wide view of resource allocation (including CPU, other general resources, and even time) rather than focusing myopically on only what the next operation is to be.

- all resources are addressed
- since the approach is general the assumptions made and the actual system can match very well, e.g., task sets can be quite complicated with periodic and aperiodic tasks, precedence constraints, value requirements, fault tolerance requirements, etc.
- the run time system can support the model accurately
- the degree of understandability is higher than the on-demand model but not as high as the static systems
- can be used for multiprocessors and distributed systems
- provides early notification that a task may miss its deadline
- highly flexible
- highest value work is executed on overload

Disadvantages include:

- no simple formula for analysis, but this does not mean that analysis cannot be accomplished; deterministic analysis can be done for critical tasks, stochastic analysis for the others, and at each point in time the system is actually performing an on-line analysis of the capabilities of the system (this is a major advantage with respect to the robustness of the system)

- the execution time cost of planning may be too high (although the cost of not having on-line monitoring and planning may be even higher in terms of having a catastrophic failure); further, a VLSI-based scheduling and resource allocation accelerator chip should be able to make the cost acceptable
- reaction time to interrupts (which were isolated) and then creating the plan for it may be too long for some tasks; consequently, multi-level scheduling is usually necessary when using planning
- no preemption of currently executing task may be too long for some required reaction times
- reserve resources even if not used; but the negative effects of this can be partially reclaimed in various ways

Static Table, Cyclic and Time Driven Systems: Real-time systems built using table, cyclic, time driven or approaches such as in Cy-clone [4] are highly predictable[2]. Each of these are based on different underlying models but they all avoid blocking and either eliminate or reduce the effects of interrupts. As pointed out in [4] when sufficient excess resources are supplied for an application, these approaches can have significant advantages. In fact, all the needs for an integrated resource allocation model are met, i.e.,

- all resources are addressed,
- the assumptions made in the model and the actual system (including the environment) match to an acceptable degree,
- the run time system supports the model in a precise manner,
- it is analyzable, and
- the system is understandable because it eliminates interrupts, preemption and blocking
- handles jitter well

[2]But be careful not to equate predictable in these systems with the notion that this necessarily means that the system will work properly. This is further based on whether the many assumptions made in the design are correct such as the worst case execution time, numbers and types of faults assumed, assumptions made with respect to correlated faults, assumptions made about the dynamic behavior of the environment, etc.

- can be used in a network
- can address fault tolerance requirements including replica determinism

Some disadvantages attributed to this approach are fictitious. For example, sometimes the way resources are addressed in these methods is ad hoc. However, this is the fault of the designers, not the approach. It is not too difficult to develop good algorithms and heuristics that can help lay out a static allocation of all resources to tasks. These algorithms may be expensive in terms of execution time, but they are off-line algorithms. Examples of such algorithms include [11] and [8], and algorithms such as [7] could easily be modified to operate in a static setting.

True disadvantages arise when the environment is more demanding than a fixed system can handle or when there are too few resources, or both. See [5] for a full description of the disadvantages of static systems.

Disadvantages are:

- the execution components can be too rigid (e.g., in some cases all computation must fit within one time unit and if not the designer must start arbitrarily breaking up tasks to these rigid boundaries)
- the compositional model, i.e., how tasks related to each other are scheduled, is too rigid resulting in a difficult to modify system. However, the resulting schedule is easy to check to make sure it works, and it is easy to understand how the system is executing.
- minor changes can result in the need for complete redesigns
- if the environment does not operate as assumed, the the fixed, pre-computed operation may be very bad
- potentially extremely low utilization and high cost
- difficult to handle functions whose computation time is long with respect to the highest rate task

The non-blocking property of these static systems contributes to analyzability and understandability. I would characterize the planning mode approach as retaining the advantages of the non-blocking aspects of the static approaches, but, for flexibility, dynamically composing well defined units into achievable schedules. The units in the planning approach may vary in size and are completely natural to the computation at hand, rather than some single chosen unit which all processing must somehow conform. With respect to this aspect of the comparison between planning and the static approaches, we can use an analogy to memory management; one could say that the cyclic approach is like paging where you choose one fixed page size, and the planning approach is more like segmentation where the units are appropriate to the function being implemented.

Summary My comments are meant to emphasize that CPU scheduling is not the big problem of real-time systems, rather a main issue is integrated cpu scheduling and resource allocation. Further, resource allocation issues are far from solved and even which basic model to use is not clear. To block or not to block, that is the question, and if you block, when to block is a subsequent question. Solutions to this problem should not be divorced for the overall system goals of predictability, meeting timing and reliability requirements, low overall system cost, and understandability. Finally, while each of the above approaches have particular real-time applications for which they are best suited, my bias is that any solution for complex real-time systems (which implicitly has certain types of requirements including operating in difficult to control environments) at least so far, based on the on-demand model produces too complicated blocking patterns that have to be too pessimistically constrained, so that for scaling and for complex tasks, it is not efficient, and that solutions based on the static approaches are not flexible enough to operate robustly in difficult to control environments. I believe that for complex real-time systems we need hybrid approaches, such as the planning approach, or other variations of it such as [1] where some degree of non-blocking has been added to the priority ceiling protocol.

References

[1] T. Baker, "Stack Based Scheduling of Real-Time Processing," *Real-Time Systems Journal*, Vol. 3, No. 1, March 1991, pp. 67-100.

[2] O. Gudmundsson, D. Mosse, K. Ko, A. Agrawala, and S. Tripathi, "MARUTI, An Environment for Hard Real-Time Applications," in *Mission Critical Operating Systems*, IOS Press, Amsterdam, 1992, pp. 75-85.

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

[4] H. Lawson, "Cy-Clone: An Approach to the Engineering of Resource Adequate Cyclic Real-Time Systems," *Real-Time Systems*, Vol. 4, 1992, pp. 55-83.

[5] C. Locke, "Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives," *Real-Time Systems*, Vol. 4, 1992, pp. 37-53.

[6] R. Rajkumar, *Synchronization in Real-Time Systems, A Priority Inheritance Approach*, Kluwer Academic Publishers, 1991.

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

[8] K. Ramamritham, "Allocation and Scheduling of Precedence Related Periodic Tasks," TR, Univ. of Massachusetts, Jan. 1992.

[9] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, Vol. 39, No. 9, Sept. 1990, pp. 1175-1185.

[10] J. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Hard Real-Time Operating Systems," *IEEE Software*, Vol. 8, No. 3, May 1991, pp. 62-72.

[11] J. Xu and D. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, March 1990, pp. 360-369.