A New Generalized Approach to Application-Defined Scheduling

Mario Aldea Rivas and Michael González Harbour

Departamento de Electrónica y Computadores
Universidad de Cantabria
39005-Santander, SPAIN
{aldeam,mgh}@unican.es

***Abstract**: In previous papers we had presented an application program interface (API) that enabled applications to use application-defined scheduling algorithms in a way compatible with the scheduling model defined in POSIX. This paper presents a new more general API with three main enhancements. First, the application scheduler is described as an abstract object that needs not be implemented as a thread, thus potentially increasing efficiency in some operating system architectures. Second, we define an abstract notion of urgency that is used by the kernel to order the threads in the scheduling queue, freeing the application scheduler of the responsibility of keeping the desired ordering of threads, and thus simplifying it and reducing overhead. In third place, we also consider thread synchronization through mutexes by adding the Stack Resource Policy or the Priority Inheritance Policy adapted to the generalized concept of urgency, both of which can be used in a large variety of fixed and dynamic priority scheduling policies without explicit intervention of the application scheduler.*

1. Introduction[1]

Most commercial real-time operating systems follow the Real-Time POSIX [2][3] standard and offer just fixed-priority scheduling as the mechanism to support real-time concurrent applications. However, it would be desirable to have dynamic priority scheduling policies available because they allow a better usage of the available processing resources. The main problem that we find is that the amount of such policies described in the literature is quite large and it is impractical for OS vendors to provide all of these policies in their implementations.

In the past years we have been working on application program interfaces (APIs) and implementations of application-defined scheduling services that could be used in operating systems following the POSIX standard [5]. Those APIs allowed the application to install one or more thread schedulers implemented as special threads, and assigning application threads to these schedulers. Although the approach presented in [5] is flexible enough to support many kinds of application defined scheduling policies, it has several drawbacks that potentially limit its efficiency:

- Because the application scheduler is a special thread, every scheduling decision requires a double context switch that in some OS architectures may be too expensive.
- The application scheduler has to keep the ordering of the threads that are ready for execution, instead of leaving this task to the underlying kernel where it can be made in a more efficient way.
- Mutual exclusive synchronization requires the intervention of the scheduler thread, and thus introduces at least two double context switches for each synchronization operation, introducing a lot of overhead.

Although the overheads measured in our MaRTE OS [1] implementation were acceptable for common applications, we realized that they could be too high in other OS architectures, and thus we worked towards eliminating these sources of inefficiency. As a result we have created a fully new API, with similar capabilities, but which is potentially much more general and efficient than the previous approach. This paper presents the model of the new approach. The API will be available with MaRTE OS [8].

2. Overview of the new scheduling model

Figure 1 shows the proposed approach for application-defined scheduling. We use a hierarchical scheduling architecture with the underlying kernel scheduler underneath, and one or more application schedulers on top.The underlying scheduler uses the thread's fixed priority as the main scheduling parameter.

Each application scheduler is a special software module that can be implemented in several ways (see Section 3), and that is responsible of scheduling a set of threads that have been attached to it.

Implementations of the application schedulers are given the freedom to choose where to place the schedulers: inside the kernel for efficiency, in the user address space for isolating the system form misbehaved schedulers, or even in a separate address space for further enhancing isolation.

In our new framework, an application scheduler has the structure shown in Figure 2. Application schedulers contain a set of operations that are invoked by the operating system each time an application-scheduled thread executes one of

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grant TIC 2002-04123-C03 (TRECOM project) and by the *Commission of the European Communities* under contract IST-2001-34140 (FIRST project)

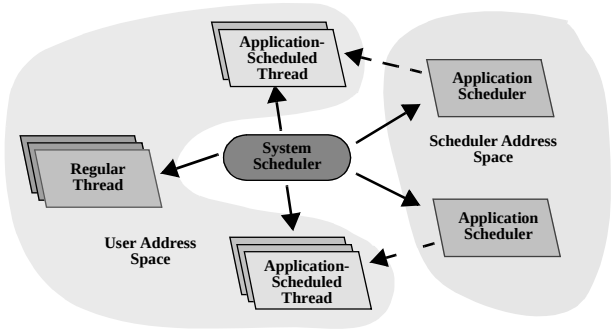

Figure 1. Model for Application Scheduling

the following actions or experiences one of the following situations which we call scheduling events:

- when a thread requests attachment to the scheduler or terminates
- when a thread blocks or gets ready
- when a thread changes its scheduling parameters
- when a thread invokes the yield operation
- when a thread explicitly invokes the scheduler
- when a thread inherits or uninherits a priority, due to the use of a system mutex

In addition to these situations, application scheduling events are also generated due to other reasons not directly caused by scheduled threads:

- when a timeout expires
- when a signal is generated for the scheduler
- when a timed notification associated with a specific thread arrives
- when a previous operation failed

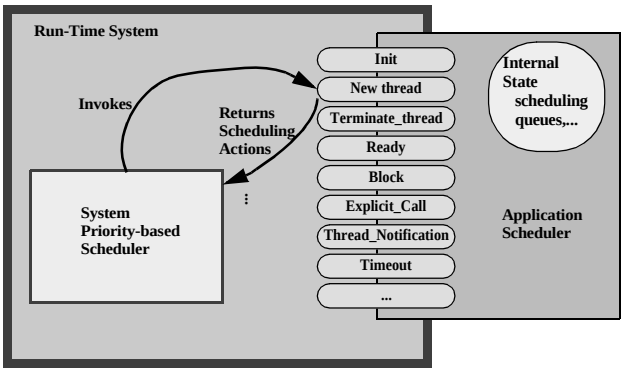

Figure 2. Structure of an Application Scheduler

A queue of scheduler events must be used by the implementation since there could be more than one pending scheduling event by the time the scheduler thread is chosen to execute. The system must ensure that the application scheduler operations are always executed before any of the application-scheduled threads attached to their scheduler, so they always take precedence over their scheduled threads. The system must also ensure that the execution of the scheduling actions and the invocation of the scheduler primitive operations are all sequential.

3. Implementations

The abstract interface presented in this paper for application-defined schedulers does not impose any particular implementation. There are several implementation possibilities, three of which are explained below.

The scheduler primitive operations could be executed in the operating system context as soon as the corresponding event is generated. So it would be executed in a non-preemptible section at the maximum priority. This mechanism could be very easy to implement in small kernels, although it has the drawback that every application scheduler has impact on all the threads in the system.

Another possibility could be to use an special thread that is activated at the appropriate priority. This implementation has the advantage that the execution of application schedulers only affects lower priority threads. As a disadvantage, efficiency may be jeopardized due to the numerous context switches between the scheduler and its scheduled threads.

An alternative solution that has the advantages of both implementations explained above is to execute primitive operations in the context of scheduled threads. To serialize execution of primitive operations they would be protected by a mutex. When an application scheduled thread is chosen by the system to execute it invokes all pending primitive operations for its scheduler. Events associated to the execution of that thread are also executed by it. A service thread is also necessary to process scheduling events in case there are no active scheduled threads at the current priority level, possibly inherited, of the scheduler. With this mechanism the scheduler only interferes with threads of priority lower than the priorities of the service thread and the scheduled threads; and the number of context switches is greatly reduced because most of the events can be handled in the context of the executing application scheduled threads. This is the implementation that is being developed in MaRTE OS.

4. Abstract Ordering of the Ready Queue

In our previous proposal for application-defined scheduling [5] the application schedulers were responsible for ordering the threads that were ready. The ready queue already existing inside the kernel was not usually practical because for a given priority level it was just a FIFO queue, and dynamic priority scheduling policies usually require other orderings. In this paper we propose a generalized

approach, similar to the one proposed by Burns [6], in which we can make use of the kernel's ready queue to order tasks based on an abstract notion of "urgency" on to which any particular scheduling parameter that the application scheduler chooses (e.g., deadline, laxity, value, quality of service, ...) can be mapped. This approach is especially suitable for scheduling policies such as EDF, in which the urgency or priority of the thread only changes from one job to the next, but remains constant within a specific job.

Each thread job is assigned a particular arithmetic value of "urgency". We have chosen a 64 bit unsigned integer representation because it can be directly mapped to the POSIX timespec time value or to the internal representation of time of most operating systems. The kernel uses this urgency number to order the tasks in the ready queue at a particular priority level, using a higher value as an indication of higher urgency. This makes it easier to implement scheduling algorithms and more important, they become more efficient because when a task finishes its current job it is not necessary to invoke the application scheduler again to determine the next task to execute. The kernel can choose the new task by itself. In this context only when a new job arrives it would be necessary to invoke the application scheduler.

5. Mutual Exclusion Synchronization

In this section we integrate two classic synchronization protocols into the application scheduling framework. Both can address mutual exclusive synchronization for many fixed- and dynamic-priority scheduling policies with bounded blocking times. The same reasons that caused POSIX to provide both priority inheritance and priority ceiling synchronization protocols are applicable to our framework and the two protocols chosen: the stack resource policy (SRP) and urgency inheritance. The priority ceiling protocol (and the SRP) has shorter worst-case blocking, is applicable to multiprocessors, and in single-processor systems it prevents deadlocks if critical sections do not suspend and also prevents context switches due to synchronization. The priority or urgency inheritance protocol have none of these properties but is still able to bound the blocking time and does not require specifying a priority ceiling for each mutex or shared resource. It is more adequate for dynamic systems and for the construction of independent software components.

5.1. Stack Resource Policy

Baker presents in [4] the Stack Resource Policy (SRP) for bounding priority inversion in real time systems, independently from the scheduling policy used. The method can be applied to fixed priority or EDF schedulers, for instance. A number called the preemption level is assigned to each thread, using the priority or importance of each thread: the higher the priority, the higher the preemption level. Shared resources are also assigned a preemption level that is the highest of the preemption levels of all the threads that may use that resource. And a new scheduling rule is imposed: a thread can only become ready for execution if its preemption level is strictly higher than the preemption levels of the resources currently locked in the system.

To use the "Stack Resource Policy" synchronization protocol we need to define a new attribute of type unsigned integer, called `preemptionlevel`, both for threads and for mutexes of protocol `PTHREAD_PRIO-_PROTECT`. In order to make the SRP compatible with the other pre-existing policies, the priority of the thread and the priority ceiling of the mutex continues to be used, and is given more weight than the new `preemptionlevel` attribute.This is equivalent to defining the actual preemption level as:

$$actual = prio \times 2^n + level$$

where n is the number of bits used to represent the preemption level. The default level for the preemption level is the maximum possible value (2^n-1). This allows threads scheduled under other policies to continue working with their usual fixed-priority scheduling rules.

For the SRP to work a new condition has to be met represented by the system ceiling rule mentioned above. Consequently, ordering threads by "urgency" without considering their preemption levels could break this SRP rule.

To correctly implement the SRP the system will keep track of the preemption levels inherited by the threads, so that when a thread owns a `PTHREAD_PRIO_PROTECT` mutex, it inherits its preemption level. The "activation" action will place the thread in the ready queue for its active priority using the "urgency" value for ordering the task, but with the additional rule that a thread cannot be inserted before any thread with higher or equal inherited preemption level. For example, in Figure 3 we can see how task T2 is less urgent than T3, but is placed before the latter in the ready queue because it has an inherited preemption level of 5. When adding T5 to the queue, we place it between T2 and T3 because, although it has an urgency of 35, its preemption level is not strictly higher than the inherited preemption level of T2.

There is one issue that requires careful attention, which is the interaction between the SRP and some server algorithms that keep track of the execution budget of a task, to make a scheduling decision when this budget is exhausted. Examples of such server algorithms are the sporadic server for fixed priorities, or the constant bandwidth server for

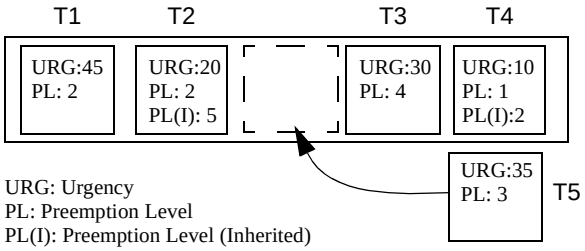

Figure 3. Rules for combining SRP and Urgency

EDF. If the thread whose budget gets exhausted is holding an SRP mutex we could break the scheduling rules if we suspend the thread. But we can notice that it is not necessary to suspend the thread, but just to change its urgency value. In this case, the thread will continue in the same place of the ready queue for its priority, because it continues to inherit the preemption level that keeps it in that position of the queue. Once the critical section finishes the thread would uninherit the mutex's preemption level, and would be rescheduled under its new urgency. This behavior allows us to avoid having to defer the scheduling events occurring while holding an SRP mutex.

5.2. Priority Inheritance

Another real-time synchronization protocol that we can use in our application-defined scheduling framework is the original priority inheritance protocol (PIP) defined by Sha, Rajkumar, and Lehoczky [7]. Although initially intended for fixed priority systems, the proofs are based on the concept of a job priority and thus can be applied to any scheduling policy, such as EDF, in which the priority of each task job is fixed, even if it varies dynamically between one job and the next.

In our particular case of application-defined scheduling the underlying scheduler is using a combination of two parameters to schedule the threads: a fixed priority and the urgency value. If we make a thread executing a critical section inherit these two values from the threads it is blocking, we can use the properties of the PIP. We therefore suggest using this protocol for application-defined scheduled threads that are using `POSIX_PRIO_INHERIT` mutexes.

The interactions of budget servers such as the SS and CBS with the priority & urgency inheritance are also safe for this protocol. If the execution budget of a particular thread *T* expires while inside a critical section with this protocol, the application scheduler may change its urgency and perhaps cause a preemption effect; but if some other thread is blocked or will become blocked by thread *T*, the latter will automatically inherit the urgency of the blocked thread and the critical section will be allowed to be completed.

6. Conclusions

In this paper we have shown a new API for a set of application defined scheduling services that can be implemented in a real-time POSIX operating system. The services allow applications to install their own schedulers implementing many kinds of scheduling policies, and schedule regular threads with these schedulers. Three fundamental aspects of application-defined scheduling are new in the proposal presented in this paper. In first place, the scheduler can be implemented in many different ways, and is no longer required for it to be executed by a special thread. Second, an abstract notion of urgency is defined to enhance efficiency and simplify the development of application schedulers. In third place, we have incorporated the stack resource policy and priority inheritance as two complementary synchronization protocols that can be used for many different kinds of schedulers, and that are capable of working correctly in the application-defined framework.

With these enhancements it is possible to develop and implement application-defined schedulers using a well defined API that will enable portability of the schedulers among different platforms.

References

[1] M. Aldea and M. González. "MaRTE OS: An Ada Kernel for Real-Time Embedded Applications". Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe-2001, Leuven, Belgium, *Lecture Notes in Computer Science, LNCS 2043*, May, 2001.

[2] ISO/IEC 9945-1:2003. *Standard for Information Technology - Portable Operating System Interface (POSIX).*

[3] IEEE Std. 1003.13-2003. *Information Technology - Standardized Application Environment Profile- POSIX Realtime and Embedded Application Support (AEP).* The Institute of Electrical and Electronics Engineers.

[4] Baker T.P., "Stack-Based Scheduling of Realtime Processes", Journal of Real-Time Systems, Volume 3, Issue 1 (March 1991), pp. 67–99.

[5] Mario Aldea Rivas and Michael González Harbour. "POSIX-Compatible Application-Defined Scheduling in MaRTE OS" Proceedings of 14th Euromicro Conference on Real-Time Systems, Vienna, Austria, IEEE Computer Society Press, pp. 67-75, June 2002.

[6] A. Burns. "Support for Deadlines and Earliest Deadline First Scheduling". Ada Issue AI95-00357-01/02. http://www.ada-auth.org/~acats/ais.html

[7] Sha, L., Rajkumar, R. and Lehoczky, J.P., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". IEEE Transaction on Computers, vol. 39, no. 9, pp. 1175-1185, September 1990.

[8] MaRTE OS home page: `http://marte.unican.es`