POSIX-Compatible Application-Defined Scheduling in MaRTE OS

By: Mario Aldea Rivas and Michael González Harbour
Departamento de Electrónica y Computadores, Universidad de Cantabria
39005 - Santander, SPAIN
{aldeam, mgh}@unican.es

Abstract[1]

This paper presents an application program interface (API) that enables applications to use application-defined scheduling algorithms in a way compatible with the scheduling algorithms defined in POSIX. Several application-defined schedulers, implemented as special user tasks, can coexist in the system in a predictable way. This API is being tested in our operating system MaRTE with the aim of proposing it to be included in a future revision of the POSIX standard.

1. Introduction

The fixed priority scheduling policies defined in the current version of the Real-Time POSIX [1] standard provide a nice combination of simplicity, predictability, and efficiency, that make them suitable for most real-time applications. However, it is well known that with dynamic priority scheduling policies it is possible to achieve higher utilization levels of the system resources than with fixed priority policies. In addition, there are many systems for which their dynamic nature make it necessary to have very flexible scheduling mechanisms, such as multimedia systems, in which different quality of service measures need to be traded against one another.

It could be possible to incorporate into the POSIX standard new dynamic scheduling policies to be used in addition to the existing policies. The main problem is that the variety of these policies is so great that it would be difficult to standardize on just a few. Different applications needs would require different policies. Instead, in this paper we propose defining an interface for application-defined schedulers that could be used to implement a large variety of scheduling policies.

The idea of application-defined scheduling has been used in many systems. A common approach is to implement the application algorithms as modules to be included or linked with the kernel (S.Ha.R.K, RT-Linux [2], Vassal [3]) this mechanism implies that the application-defined scheduling algorithm will be executed inside the kernel. Another solution is proposed in the CPU Inheritance Scheduling [4], in which the kernel only implements thread blocking, unblocking and CPU donation, and the application defined schedulers are tasks which donate the CPU to other tasks. In this approach the only method used to avoid priority inversion is the CPU inheritance. A different approach is followed by RED-Linux [5]: a two-level scheduler is used, where the upper level is implemented as a user process that maps QOS parameters into low-level attributes to be handled by the lower level scheduler. With that mechanism many scheduling algorithms can be implemented although it is not general and protocols for shared resources are not addressed.

Introducing application-defined scheduling in POSIX has some challenges that do not appear in existing interfaces and implementations. One of the most difficult ones, is to keep the new schedulers compatible with the existing scheduling policies, while allowing implementations in which the application schedulers are not allowed to "invade" the operating system kernel space. Another one is to use as much of the existing scheduling interface as possible, adding the fewest possible new interfaces.

The interface designed is been tested in our operating system MaRTE OS [6] (Minimal Real-Time Operating System for Embedded Applications). MaRTE OS is a real-time kernel for embedded applications that follows the Minimal Real-Time POSIX.13 subset, providing both the C and Ada language POSIX interfaces. It allows cross-development of Ada and C real-time applications. Mixed Ada-C applications can also be developed, with a globally consistent scheduling of Ada tasks and C threads.

2. Requirements

The following requirements were stated for the application-defined scheduling interface in POSIX:

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grant TIC99-1043-C03-03

- The new scheduling policies shall have a behavior compatible with other existing scheduling policies in POSIX.
- It shall be possible to isolate critical parts of the application from failures in the application-defined schedulers.
- It should be possible to define several application-defined schedulers.
- It should be possible to execute the application-defined scheduler in an execution environment different than that of regular application threads, for example inside the kernel. But the interface should also allow the implementation to execute the scheduler in the environment of the application threads.
- The application-defined scheduler should have the ability to determine the time intervals at which the different threads scheduled under it run.
- If an application-scheduled thread needs to synchronize with other system-scheduled threads, there needs to be a portable mechanism to bound priority inversion.
- It should be possible to define application-defined protocols to access the resources.
- It should be possible for an application-scheduled thread to pass information to its scheduler.
- It should be possible to filter the specific scheduling events that the system notifies to the scheduler (for efficiency purposes).
- It should be possible to attach application-specific data to a mutex.
- It should be possible to achieve multiprocessor application scheduling. Efficient multiprocessor scheduling will require knowledge of the specific architecture, and in particular of the number or processors capable of executing application-scheduled threads simultaneously.
- Each scheduler should be capable of activating many of its scheduled threads at the same time, and/or to block previously activated tasks.
- It should be possible to have a mechanism for sharing memory among scheduler threads, their scheduled threads, other scheduler threads, and/or regular threads.

3. Model for Application-Defined Scheduling

In the proposed approach for application-defined scheduling, each application scheduler is a special kind of thread, that is responsible of scheduling a set of threads that have been attached to it. This leads to two classes of threads in this context:

- *Application scheduler threads*: special threads used to run application schedulers.
- *Regular threads*: regular application threads

According to the way a thread is scheduled, we can categorize the threads as:

- *System-scheduled threads*: these threads are scheduled directly by the operating system, without intervention of a scheduler thread.
- *Application-scheduled threads*: before they can be scheduled by the system, they need to be activated by their application-defined scheduler.

It is unspecified whether application scheduler threads can themselves be application scheduled. They can always be system scheduled.

Because mutexes may cause priority inversions, it is necessary that the scheduler thread knows about the use of mutexes to establish its own protocols, possibly different from the priority ceiling or priority protection protocols currently available in POSIX. For this purpose, two kinds of mutexes will be considered:

- *System-scheduled mutexes*. Those created with the current POSIX protocols: no priority inheritance (PTHREAD_PRIO_NONE), immediate priority ceiling (PTHREAD_PRIO_PROTECT), or basic priority inheritance (PTHREAD_PRIO_INHERIT).
- *Application-scheduled mutexes*: Those created with PTHREAD_APPSCHED_PROTOCOL. The protocol itself will be defined by the application scheduler.

3.1. Relations with other Threads

Each thread in the system, whether application- or system-scheduled, has a system priority.

For system-scheduled threads, the system priority is the priority defined in its scheduling parameters (*sched_priority* field of its *sched_param* structure), possibly modified by the inheritance of other priorities through the use of mutexes.

For application-scheduled threads, the system priority is lower or equal than the system priority of their scheduler. The system priority of an application-scheduled thread may change because of the inheritance of other system priorities through the use of mutexes. In that case, its scheduler also inherits the same system priority (but it is not inherited by the rest of the threads scheduled by that scheduler). In addition to the system priority, application-scheduled threads have application scheduling parameters that are used to schedule that thread among the other threads attached to the same application scheduler. The system priority always takes precedence over any application scheduling parameters, therefore application-scheduled threads and their scheduler take precedence over threads with lower system priority, and they are always preempted by threads with higher system priority that become ready. The scheduler always takes precedence over its scheduler threads.

If application-scheduled threads coexist at the same priority level with other system-scheduled threads, then the POSIX scheduling rules apply as if the application-scheduled threads were scheduled under the FIFO within priorities policy (SCHED_FIFO); so another SCHED_FIFO thread runs until completion, until blocked, or until preempted, whatever happens earlier. A thread running under the round-robin within priorities policy (SCHED_RR) runs until completion, until blocked, until preempted, or until its round robin quantum has been consumed, whatever happens earlier. Of course, because the interactions between the different policies may be difficult to analyze, the normal use will be to have the scheduler thread and its scheduled threads running at an exclusive system priority level.

In the presence of priority inheritance, the scheduler inherits the same priorities as its scheduled tasks, to prevent priority inversions from occurring. This means that high priority tasks that share mutexes with lower system priority application threads must take into account the scheduler overhead when accounting for their blocking times.

3.2. Relations Between the Scheduler and its Attached Threads

Each application-defined scheduler may activate many application-scheduled threads to run concurrently. The scheduler may also block previously activated threads. Among themselves, concurrently scheduled threads are activated like SCHED_FIFO threads. As mentioned previously, the scheduler always takes precedence over its scheduled threads.

For an application-scheduled thread to become ready it is necessary that its scheduler activates it. When the application thread executes one of the following actions or suffers one of the following events a scheduling event is generated for the scheduler, unless the scheduling event to be generated is being filtered out (discarded).

- when the thread is created attached to the scheduler
- when the thread blocks
- when the thread changes its scheduling parameters
- when a thread invokes the yield operation
- when a thread explicitly invokes the scheduler

The application scheduler is a special thread whose code is usually a loop where it waits for a scheduling event to be notified to it by the system, and then determines the next application thread to be activated.

The scheduler being a single thread implies that its actions are all sequential. For multiprocessor systems this may seem to be a limitation, but for these systems several schedulers could be running simultaneously on different processors, cooperating with each other. For single processor systems the sequential nature of the scheduler should be no problem. Again, it is possible to have several scheduler threads running at the same time, and cooperating with each other by synchronizing through regular mutexes and condition variables.

4. Activation and Suspension of Application-Scheduled Threads

The main point in our interface is the *posix_appsched_execute_actions*() function, which allows the application scheduler to execute a list of scheduling actions. Each element of that list will cause an application-scheduled thread to be activated or suspended. After the execution of the scheduling actions the calling scheduler thread shall suspend waiting for the next scheduling event.

If desired, a timeout can be set as an additional return condition which will occur when there is no scheduling event available but the timeout expires.

The time measured immediately before the function returns can be requested if it is relevant for the algorithm.

The C language prototype of this function is:

```
int posix_appsched_execute_actions (
   const posix_appsched_actions_t *sched_actions,
   const struct timespec *timeout,
   struct timespec *current_time,
   struct posix_appsched_event *event);
```

5. Schedulings Events

The scheduling events are stored in a FIFO queue until processed by the scheduler. Each event carries the following information with it:

- Event code
- Thread that caused the event
- Additional information associated with the event
  - Inherited (or uninherited) system priority
  - Timestamp when the thread was preempted
  - Pointer to an application-scheduled mutex
  - Specific information

The specific events that may be notified to the scheduler thread are shown in Table 1.

6. Current Status and Further Work

With the purpose of proving the suitability of our interface it is being implemented and tested in MaRTE OS. Several scheduling algorithms not defined in the POSIX standard are been implemented, such as EDF, the Constant Bandwidth Server, Proportional Share Algorithm, etc.

Another important aspect to be measured is the overhead introduced by our interface. With this aim, the Sporadic

Table 1. Scheduling Events

Event Code	Description	Additional information
POSIX_APPSCHED_NEW	New thread created	NULL pointer
POSIX_APPSCHED_TERMINATE	A thread has been terminated	NULL pointer
POSIX_APPSCHED_READY	A thread has become unblocked by the system	NULL pointer
POSIX_APPSCHED_BLOCK	A thread has blocked	NULL pointer
POSIX_APPSCHED_YIELD	A thread has invoked *pthread_yield*()	NULL pointer
POSIX_APPSCHED_PREEMPT	A thread was preempted by a thread with a higher system priority.	Timestamp or a zero time value
POSIX_APPSCHED_CHANGE_SCHED_PARAM	A thread has changed its scheduling parameters	NULL pointer
POSIX_APPSCHED_EXPLICIT_CALL	A thread has explicitly invoked the scheduler	Message
POSIX_APPSCHED_TIMEOUT	A timeout has expired	NULL pointer
POSIX_APPSCHED_PRIORITY_INHERINT	A thread has inherited a new system priority due to the use of system mutexes	Inherited system priority
POSIX_APPSCHED_PRIORITY_UNINHERIT	A thread has finished the inheritance of a system priority	Uninherited system priority
POSIX_APPSCHED_INIT_MUTEX	A thread has initialized an application-scheduled mutex	Pointer to the mutex
POSIX_APPSCHED_DESTROY_MUTEX	A thread has destroyed an application-scheduled mutex	Pointer to the mutex
POSIX_APPSCHED_LOCK_MUTEX	A thread has acquired the lock of an application- scheduled mutex	Pointer to the mutex
POSIX_APPSCHED_UNLOCK_MUTEX	A thread has released the lock of an application-scheduled mutex	Pointer to the mutex
POSIX_APPSCHED_BLOCK_AT_MUTEX	A thread has blocked at an application-scheduled mutex	Pointer to the mutex

Server Algorithm will be implemented as an application-defined policy in order to be compared with the Sporadic Server Algorithm implemented inside the kernel. The overhead introduced for the interface to the POSIX defined policies, even in the absence of any application scheduler, is going to be measured as well.

For the purpose of standardization, the application-defined scheduling API will be proposed for the next revision of the POSIX standard; in addition, a POSIX-Ada interface will be developed.

References

[1] ISO/IEC 9945-1 (1996). *ISO/IEC Standard 9945-1:1996. Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) [C Language]*. Institute of Electrical and electronic Engineers.

[2] Yodaiken V., "An RT-Linux Manifesto". Proceedings of the 5th Linux Expo, Raleigh, North Carolina, USA, May 1999.

[3] George M. Candea and Michael B. Jones, "Vassal: Loadable Scheduler Support for Multi-Policy Scheduling". Proceedings of the Second USENIX Windows NT Symposium, Seattle, Washington, August 1998.

[4] Bryan Ford and Sai Susarla, "CPU Inheritance Scheduling". Proceedings of OSDI, October 1996.

[5] Y.C. Wang and K.J. Lin, "Implementing a general real-time scheduling framework in the red-linux real-time kernel". Proceedings of IEEE Real-Time Systems Symposium, Phoenix, December 1999.

[6] Mario Aldea Rivas and Michael González Harbour, "MaRTE OS: An Ada Kernel for Real-Time Embedded Applications". To appear, Ada-Europe-2001, Lovaina, Bélgica, May 2001.