A Round Robin Scheduling Policy for Ada

A. Burns[1], M. González Harbour[2], and A.J. Wellings[1]

[1] Department of Computer Science
University of York, UK
Email: {burns,andy}@cs.york.ac.uk
[2] Departamento de Electrónica y Computadores
Universidad de Cantabria, Spain
Email: mgh@unican.es

Abstract. Although Ada defines a number of mechanisms for specifying scheduling policies, only one, `Fifo_Within_Priorities` is guaranteed to be supported by all implementations of the Real-Time Systems Annex. Many applications have a mixture of real-time and non real-time activities. The natural way of scheduling non real-time activities is by time sharing the processor using Round Robin Scheduling. Currently, the only way of achieving this is by incorporating yield operations in the code. This is ad hoc and intrusive. The paper proposes a new scheduling policy which allows one or more priority levels to be identified as round robin priorities. A task whose base priority is set to one of these levels is scheduled in a round robin manner with a user-definable quantum.

1 Introduction

The Real-Time Systems Annex in Ada 95 defines a number of mechanisms for specifying scheduling policies. It also provides a complete definition of one such policy: `Fifo_Within_Priorities`. This policy, which requires preemptive priority based dispatching, is a natural choice for real-time applications. It can be implemented efficiently and leads to the development of applications that are amenable to effective analysis – especially when combined with the immediate priority ceiling protocol (ceiling locking) on protected objects.

There are, however, application requirements that cannot be fully accomplished with this policy alone. For example, many applications have a mixture of real-time and non real-time activities. The natural way of scheduling non real-time activities is by time sharing the processor, as in most general-purpose operating systems. With the standard policy (`Fifo_Within_Priorities`), some level of rotation can be achieved by giving the set of non real-time tasks the same priority and requiring them to incorporate periodic yield operations (such as delay 0.0). However this is an ad hoc approach, is intrusive, and cannot easily be undertaken with legacy code or when using prewritten or shared libraries. Real-time applications can also benefit from a round robin approach. Although extra task switches increases run-time overheads, round robin execution allows a set of tasks (with the same priority) to make progress at a similar rate.

The aim of this paper is to define a new scheduling policy for the Real-Time Systems Annex, with a view to this definition (or one derived from it) being incorporated into

the Annex. This would be one of a set of new policies being proposed for the Annex. For example, a non-preemptive dispatching policy has already been agreed[1].

This paper is organised as follows. First an overview of the POSIX provision is given. Ada is closely associated with POSIX and indeed some Ada run-time systems (e.g. GNAT) are built on top of POSIX compliant kernels. It is important that any proposal for Ada is implementable on POSIX even if the details of the Ada policy are not identical to the Round Robin facility of POSIX. Section 3 then gives the requirements of the Ada policy and Section 4 the details of the proposal (including a discussion of some of the priority inheritance problems that ensue). Conclusions are given in Section 5.

2 The POSIX Policies

The POSIX real-time scheduling model [3, 2] is a fixed-priority preemptive model in which there are three compatible scheduling policies defined:

- SCHED_FIFO. It is a priority preemptive policy that uses FIFO ordering for threads of the same priority. It is similar to Ada's FIFO_Within_Priorities policy.
- SCHED_RR. It is also a priority-preemptive policy that uses a round-robin execution quantum to share the processor among threads of the same priority level. In this policy, when the implementation detects that a running thread has been executing for a time period of the round robin quantum or longer, the thread is placed at the tail of the scheduling queue for its priority, and the head of that queue is removed and made the running thread. While a round robin thread is preempted by higher priority threads it does not consume its unused portion of round robin quantum. This policy ensures that if there are multiple threads at the same priority, one of them will not monopolize the processor.
- SCHED_SS. It is the sporadic server scheduling policy, intended for processing aperiodic threads with a guaranteed bandwidth, and with predictable effects over lower priority threads.

These policies can be set on a thread by thread basis, because the effects of mixing them are well defined. For example, when a round robin thread is running, its execution time is limited to its time quantum. After the quantum is elapsed, the thread is sent to the tail of the ready queue for its priority. If a FIFO within priorities thread now comes into execution, it runs until completion (possibly preempted by higher priority threads during its execution). It is the responsibility of the application developer to make sure that no mixture of round robin and FIFO threads is made at the same priority level, if the round robin semantics is to be preserved.

The ranges of valid priorities for the SCHED_FIFO and SCHED_RR may coincide, overlap, or be disjoint. Each of these ranges is required to have at least 32 distinct priority levels. There are functions that allow the application to obtain such priority ranges in a portable way.

In order to portably define a set of priorities that is suitable for mixing real-time and non real-time threads, a new requirement is being proposed in the revision of the POSIX.13 real-time profiles [4], which in essence states that there should be at least one

round robin priority level that is below the first 31 priority levels of the SCHED_FIFO policy. In this way, in a portable application, the non real-time threads would use that round robin priority level, while the real-time threads would use the top 31 values of the SCHED_FIFO policy.

Real-time POSIX defines the mutex as the mechanism to achieve mutual exclusive synchronization for shared data and resources. Mutexes have an optional creation attribute, the protocol, that can be used by the application to specify a real-time synchronization protocol, if desired. Two such protocols are defined:

- PTHREAD_PRIO_INHERIT: This is the basic priority inheritance protocol, in which a thread that is blocking one or more threads due to the use of the mutex inherits their priorities.
- PTHREAD_PRIO_PROTECT: This is the immediate priority ceiling protocol, in which each mutex has an attribute called its priority ceiling; while a thread holds the lock on one or more mutexes, it inherits the priority ceilings of those mutexes. This is basically the same as Ada's Ceiling_Locking protocol for protected objects.

One problem that affects real-time behavior is the relationship between the round robin scheduler, and the mutexes. Ideally, to minimize blocking, expiration of the round robin quantum should be delayed if the thread is holding a mutex with one of the PTHREAD_ PRIO_INHERIT of PTHREAD_PRIO_PROTECT protocols. The standard does state that "While a thread is holding a mutex that has been initialized with the PRIO_INHERIT or PRIO_PROTECT protocol attributes, it shall not be subject to being moved to the tail of the scheduling queue at its priority". But the standard does not list the expiration of the round robin quantum as one of the circumstances to which this statement applies, thus leaving the behavior unspecified.

2.1 Overheads of Implementation

Depending on the degree of precision that the implementtaion gives to round robin scheduling, the execution time quantum can be measured in a coarse way, for example counting ticks of a particular clock (and accounting for partial ticks as full ones), or in a more precise way by accounting for actual execution time.

As an example, we have implemented a precise round robin scheduler in MaRTE OS [6], and we have measured its overhead relative to the overhead of the SCHED_FIFO policy. We have mixed the implementation of this scheduler with the execution-time budget mechanism also available in MaRTE OS. For example, the expirations of round robin quantum are programmed as execution time events, similar to those of other budgets associated with execution-time timers. The implementation of execution time budgets has taken approximately 66 lines of code, and has incremented the standard context switch from 0.42 microseconds to 0.44, a negligible 20 ns increase (as measured on a 1.1GHz Pentium III). The implementation of the round robin policy itself has taken 30 lines of code, of which 10 lines are shared with the sporadic server scheduling policy, also specified in POSIX and available in MaRTE OS. The context switch time due to the exhaustion of a round robin quantum, 0.75 us, is very similar to the context switch caused by the expiration of a relative sleep algorithm, 0.75 us. Both include the time

required to handle the timer hardware interrupt, which was not necessary for the regular context switch. Although these numbers are those inside the OS and do not take into account the time spent by the Ada Runtime System, it is expected that the overheads there will be comparable.

3 Requirements for Round Robin Scheduling

Although the basic requirements for Round Robin Scheduling are straightforward, and many examples exist in general purpose operating systems (as well as the POSIX interface definition), there is more than one approach possible to providing this dispatching behaviour; see Aldea Rivas and González Harbour for a discussion [5]. Clearly a non-intrusive method is needed. Hence a policy must be defined that forces Round Robin scheduling on a set of tasks without the need to make any code changes to the tasks themselves. One approach, following POSIX, would be to assign Round Robin status on a per task basis. This is a very general facility that is appropriate for an OS interface definition. However, from the perspective of the Ada Language, introducing a per-task scheduling policy represents a substantially different model than the one currently defined in the Real-Time Annex. In addition, allowing a mixture of tasks with Round_Robin and FIFO_Within_Priorities at the same priority level introduces a potential for mistakes that would degrade real-time performance. The following requirements are therefore necessary and sufficient.

- Round Robin (RR) scheduling should co-exist with standard preemptive priority based scheduling.
- All tasks at any priority level should be dispatched according to a single policy (either `Fifo_Within_Priorities` or a round robin).
- At a RR priority level, each task is given the same quantum of CPU resource.

This different uses of priority are illustrated in Figure 1.

When a task exhausts its quantum of CPU resource, it is placed at the back of the dispatching queue of tasks at that priority level.

Hence, we take a priority-centred rather than a task-centred view. A full proposal (see Section 4) must, therefore, cater for the use of dynamic priorities that may move a task into or out of a RR priority level. It must also give a meaning to a protected object being assigned a RR priority.

A significant requirement is that the proposal must deal with the situation in which a task's quantum is exhausted while it has an inherited priority. In particular

- No extra lock must be needed to ensure mutual exclusive execution with protected objects.

The most straightforward use of Round Robin scheduling is to require it only at the lowest priority level (as with the new POSIX proposal [4]). All other priorities use the normal preemptive scheme, but the spare capacity that becomes available to the lowest priority level is shared between a set of non real-time tasks. Scheduling analysis could be used to give a minimum value to this spare capacity (over a relatively long time

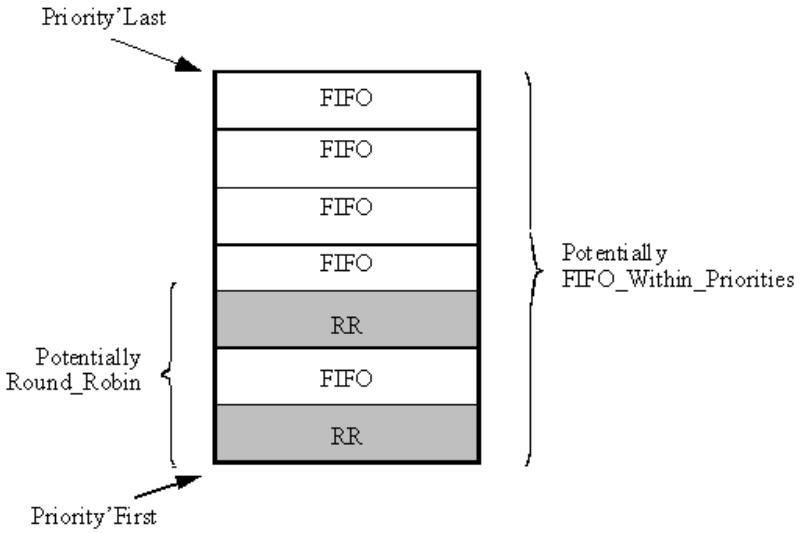

Fig. 1. Priority Levels

period) and hence, with knowledge of the quantum size and number of tasks at the RR level, a measure of progress could be calculated.

Round Robin scheduling can also be used at priority levels other than the lowest one. Here, the goal is to reduce the output jitter of a set of periodic tasks which all have the same period and deadline. Without Round Robin scheduling, the variation in the scheduling of these tasks results in a large variation in their response times. If the tasks are outputting to the environment, it may be necessary to bound the jitter on that output. This use of Round Robin scheduling within a real-time program explains our proposal to place the new policy in Annex D (the real-time annex).

A final observation concerns the size of the quantum. Ideally this should be programmable (at least during elaboration) so that each RR priority level can choose the size of its quantum. However, some implementations, for example those built upon POSIX, may need to impose restrictions. A compromise is therefore needed in the proposal.

In a current AI (Ada Issue) a proposal is being considered for adding CPU-Time budgeting capabilities to the Ada standard. This AI defines a new package, `Execution_Time` (as a child of `Ada.Real_Time`), that contains operations to measure the execution time of the different tasks in the system, and to limit the execution time of the tasks to desired execution time budgets, with facilities to detect execution time overruns. As part of this package, a private type called `CPU_Time` is defined to represent periods of execution time. This type seems the most appropriate for specifying the round robin quantum values, and therefore we have used it in this proposal.

4 Proposal

An extension is proposed for Annex D of the Ada Reference Manual (ARM). First a new policy-identifier is defined for task dispatching.

```
pragma Task_Dispatching_Policy(Priority_Specific);
```

When `Priority_Specific` is used, the dispatching policy is defined on a per priority level. This is achieved by the use of a new configuration pragma:

```
pragma Priority_Policy (Policy_Identifier, Priority
                        [,Policy_Argument_Definition]);
```

The `Policy_Identifier` shall be `Fifo_Within_Priorities`, `Round_Robin` or an implementation-defined identifier. For `Policy_Identifier Fifo_Within_Priority` there shall be no `Policy_Argument_Definition`. For `Policy_Identifier Round_Robin`, the `Policy_Argument_Definition` shall be a single parameter, `Round_Robin_Quantum`, of type `CPU_Time` (from the Execution Time proposal).

For other policy identifiers, the semantics of the `Policy_Argument_Definition` are implementation defined. At all priority levels, the default `Priority_Policy` is `Fifo_Within_Priorities`. The locking policy associated with the `Priority_Specific` task dispatching policy is `Ceiling_Locking`.

An implementation that supports Round Robin Scheduling must provide the following package:

```
with Ada.Real_Time.Execution_Time; use Ada.Real_Time;
package Round_Robin_Dispatching is
  Default_Quantum : constant Execution_Time.CPU_Time;
  Minimum_Quantum : constant Execution_Time.CPU_Time;
  Maximum_Quantum : constant Execution_Time.CPU_Time;
  function Nearest_Supported_Quantum
           (Q : Execution_Time.CPU_Time)
           return Execution_Time.CPU_Time;
  Maximum_Priority_Level : constant System.Priority;
private
  -- definition of constants, implementation defined
end Round_Robin_Dispatching;
```

Here a default quantum is given together with the minimum and maximum values supported. Note although the type is private, a value in seconds can be obtained via the procedure `Split` defined in `Execution_Time`. If an implementation only supports one value then the minimum and maximum are set to the default. The function is provided to deal with implementations that only support discrete values for the quantum. If the user calls this function with a desired budget then the function returns the actual value that will be used at run-time (this will be at least the minimum and no more than the maximum).

The final constant in the package gives the maximum priority that can be specified for RR dispatching. Note this is of type `Priority` and hence cannot be at an interrupt

priority level. The lowest value this can be is `Priority'First`; in which case only the lowest priority is available for RR scheduling.

Use of pragma `Priority_Policy` will be checked with the following rules applying:

- If the same priority is given in more than one pragma, the partition is rejected.
- If the `Policy_Identifier` is `Round_Robin` and the value of `Priority` is greater than `Round_Robin_Dispatching.Maximum_Priority_Level` then the partition is rejected.
- If the `Policy_Identifier` is `Round_Robin` and no quantum is given, the default in `Round_Robin_Dispatching` applies.
- If `Task_Dispatching_Policy` is not `Priority_Specific` for the partition in which a pragma `Priority_Policy` appears, then the partition is rejected.

It is now necessary to consider the dynamic semantics of the proposal. If all priority levels have `Priority_Policy Fifo_Within_Priorities` then this is equivalent to `Task_Dispatching_Policy Fifo_Within_Priorities`. The dynamic semantics defined in D.2.2 of the ARM for `Fifo_Within_Priorities` apply to any priority level with `Priority_Policy Fifo_Within_Priorities`.

For `Policy_Identifier Round_Robin`, the same rules for `Fifo_Within_Priority` apply with the additional rules and modifications:

- When a task is added to the tail of the ready queue for a priority level with `Priority_Policy Round_Robin`, it has an execution time budget set equal to the `Round_Robin_Quantum` for that priority level. This will occur when a blocked task becomes executable again.
- When a task is preempted (by a higher priority task), it is added to the head of the ready queue for its priority level. If this is a RR priority level then it retains its remaining budget.
- When a task with a base priority at a RR priority level is executing, its budget is decreased by the amount of execution time it uses.
- When the implementation detects that a task with a round robin priority has been executing for a time larger than or equal to its round-robin quantum, the task is said to have exhausted its budget. When a running task exhausts its budget, it is moved to the tail of the ready queue for that priority level. The semantics of this move is equivalent to the task with priority `Pri` executing `Set_Priority(Pri)`; see D.5(15) of the ARM. Hence, for example, it will continue to execute within a protected operation.

The last rule together gives the important details of the proposal. First it is the base priority of a task that is significant. If a task's base priority is at a RR level then it will consume its budget whenever it is executing even when it has inherited a higher priority (i.e. its active priority is greater than its base priority). The final point deals with the key question of what happens if the budget becomes exhausted while executing in a protected object. To ensure mutual exclusion, without requiring a further lock, it is necessary to allow the task to keep executing within the PO. It will consume more than its quantum but the expected behaviour of system is maintained. The usual programming

discipline of keeping the code within protected objects as short as possible will ensure that quantum overrun is minimised. Further support for these semantics comes from observing that execution within a PO is abort-deferred. Quantum exhaustion is a less severe state than being aborted; deferred behavior thus seems appropriate.

To complete the definition, a few details have to be covered. A task that has its priority changed, via the use of `Set_Priority`, may move to, or from, a round-robin priority level. If it is moved to a RR level then it is placed at the tail of the ready queue and given a full quantum. If it moves to a `Fifo_Within_Priorities` scheme then it is again placed at the tail of the ready queue but no quantum is set. Figure 2 gives an illustration of the rules defined above.

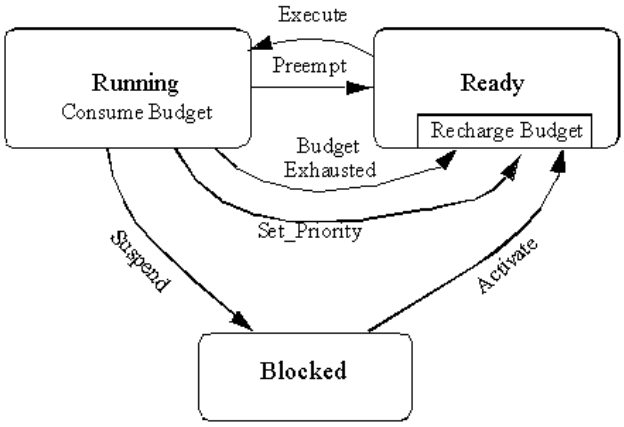

Fig. 2. State Transistion Diagram

There are no additional rules concerning a protected object with a priority assigned that is a round robin priority. A task does not obtain a budget by executing with an active priority at a round robin level – it is only the base priority of a task that determines its scheduling policy.

An example of usage is as follows. Assume package `Ada.Real_Time` is visible. To set the default priority level to round-robin with a quantum of 50 milliseconds:

```
pragma Task_Dispatching_Policy (Priority_Specific);
```

```
pragma Priority_Policy (Round_Robin, Default_Priority,
                        Execution_Time.Time_Of(0,Milliseconds(50));
```

By setting the quantum to be very large (if allowed by `Round_Robin_Dispatching.Maximum_Quantum`) a 'run until blocked' at the lowest priority level can be requested:

```
pragma Task_Dispatching_Policy (Priority_Specific);

pragma Priority_Policy (Round_Robin, Priority'First,
                        Execution_Time.CPU_Time_Last);
```

This proposal introduces round-robin scheduling by assuming that such a scheme would be used at a particular priority level (or levels) and that a single quantum value is appropriate for each level. Mixing round-robin and non-round-robin at the same priority level, or having quanta defined on a per-task basis is not considered necessary – and could be achieved by using supervisor tasks and execution time budgeting.

Increased functionality could be achieved by allowing the size of the quantum to be changed dynamically. This is not considered necessary.

4.1 Scheduling Analysis

Most forms of scheduling analysis (e.g. response-time analysis) assume that all tasks have distinct priorities. If two or more tasks share a priority then the analysis has to assume that, for each task, the other tasks execute first. This is a pessimistic but safe assumption; whatever the actual ordering, the analysis will not underestimate the interference of other tasks. Round robin dispatching is therefore already catered for by this approach, and hence standard scheduling analysis can be applied (with a small amount of extra overhead been factored in to allow for the increased number of context switches).

5 Conclusions

This paper has introduced a proposal for adding Round Robin scheduling to the portfolio of policies supported by the Real-Time Systems Annex. Although priority based preemptive scheduling is the policy of choice for most real-time applications, it is increasingly the case that large have both more complete real-time requirements and non real-time components. With just one non real-time task it is easy to construct the code of the task as a non-blocking infinite loop, and give it the lowest priority in the system. But if there are two or more such tasks it is not possible to share this base priority without injecting 'relinquish CPU' statements. Round Robin scheduling is the standard way of time sharing such tasks. Such sharing is also of benefit to real-time code by allowing a set of tasks to make progress at a similar rate.

The motivation for the proposal given here is to allow applications to simultaneously use priority based scheduling and Round Robin. It is intended to forward this proposal (or variant of it) as an AI to the ARG for possible inclusion in the next Ada revision.

Acknowledgements

The topic of Round Robin scheduling has been discussed at the Real-Time Workshops, IRTAW – we acknowledge the input from people attending these workshops. The proposal reflects work undertaken as part of the EU funded FIRST project.

References

1. A.Burns. Non-preemptive dispatching and locking policies. In M. Gonzále Harbour, editor, *Proceedings of the 10th International Real-Time Ada Workshop*, pages 46–47. ACM Ada Letters, 2001.
2. IEEE. Information Technology - Standardized Application Environment Profile - POSIX Realtime and Embedded Application Support (AEP) (POSIX): 1003.13:1998, 1998.
3. IEEE. Information Technology - Portable Operating System Interface (POSIX): 1003.1:2001, 2001.
4. IEEE. Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP) Draft 1003.13 (2), 2002.
5. M. Aldea Rivas and M. González Harbour. Extending Ada's real-time systems annex with the POSIX scheduling services. In M. González Harbour, editor, *Proceedings of the 10th International Real-Time Ada Workshop*, pages 20–26. ACM Ada Letters, 2001.
6. M. Aldea Rivas and M. González Harbour. MaRTE OS: An ada kernel for real-time embedded applications. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference, Leuven*. Springer Verlag, LNCS 2043, 2001.