AI-00298 Non-Preemptive Dispatching

!standard D.2.4 (00) 04-05-24 AI95-00298/05

!reference AI95-00321
!class amendment 02-06-01
!status Amendment 200Y 03-07-02
!status WG9 Approved 04-06-18
!status ARG Approved 7-0-1 03-02-08
!status work item 02-06-01
!status received 02-06-01
!priority Medium
!difficulty Medium
!subject Non-Preemptive Dispatching

!summary

A new dispatching policy is defined for the non-preemptive execution of Ada tasks.

!problem

The Real-Time Annex requires that a preemptive dispatching policy is always provided by an implementation. It also allows implementation-defined policies to be supplied. For many system builders, particularly in the safety-critical area, the policy of choice is non-preemption.

The standard way of implementing many high-integrity applications is with a cyclic executive. Here a sequence of procedures is called within a defined time interval. Each procedure runs to completion, there is no concept of preemption. Data is passed from one procedure to another via shared variables; no synchronization constraints are needed since the procedures never run concurrently.

There is a need for a standard way of obtaining non-preemptive execution for Ada programs whilst still using tasks as a convenient means of encapsulating concurrent entities.

!proposal

A new task dispatching policy is defined, namely Non_Preemptive_FIFO_Within_Priorities. It is one of a number of additional task dispatching policies. The proposal assumes wording changes have been agreed for sections D.2.1 and D.2.2 (see AI95-00321).

!wording

New section D.2.4:

D.2.4 Non-Preemptive Dispatching

A non-preemptive dispatching policy is defined by policy_identifier Non_Preemptive_FIFO_Within_Priorities.

Legality Rules

Non_Preemptive_FIFO_Within_Priorities can be specified as the policy_identifier of pragma Task_Dispatching_Policy (see D.2.2).

Post-Compilation Rules

If Non_Preemptive_FIFO_Within_Priorities is specified for a partition then Ceiling_Locking (see D.3) shall also be specified for that partition.

Dynamic Semantics

When Non_Preemptive_FIFO_Within_Priorities is in effect, modifications to the ready queues occur only as follows:

o When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.

o When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority.

o When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority. This is a task dispatching point (see D.2.1).

Implementation Permission

Since implementations are allowed to round all ceiling priorities in subrange System.Priority to System.Priority'Last (see D.3), an implementation may allow a task to execute within a protected object without raising its active priority provided the protected object does not contain pragma Interrupt_Priority, Interrupt_Handler or Attach_Handler.

!discussion

Three recent meetings of the International Real-Time Ada Workshop (IRTAW) considered a non-preemptive version of Ravenscar, and non-preemptive execution in general. Class A (or Class 1) software (as defined in safety standards such as DO-178B) typically has a very restricted architecture. Often only periodic behaviours need be supported. In order to reduce non-determinism and to increase the effectiveness of testing, non-preemptive execution is desirable.

The standard way of implementing many high-integrity applications is with a cyclic executive. Using this technique a sequence of procedures is called within a defined time interval. Each procedure runs to completion and there is no concept of preemption. Data is passed from one procedure to another via shared variables and no synchronization constraints are needed, since the procedures never run concurrently.

The development of the Ravenscar profile has shown how a simple subset of the tasking features of Ada can give effective support to high-integrity real-time systems. Whilst many system builders are prepared to move to use tasking (as defined by Ravenscar) some are reluctant to embrace the preemptive dispatching policy. They prefer the greater determinism of non-preemptive dispatching.

One of the advantages of non-preemption is that, on a single processor, shared data does not require any form of lock to furnish mutual exclusion - provided the task does not block within the critical section. Ada defines the code within a protected object (PO) to be free of potential suspension. Hence POs remain the means of defining critical sections of code.

The major disadvantage with non-preemption is that it will usually (although not always) lead to reduced schedulability. For example, consider a low priority (long deadline) task with a long execution time. Once it starts executing, no high priority (short deadline) task will be able even to start its execution until the low priority task has finished. This is an example of excessive priority inversion. To reduce its impact the low priority task needs to periodically offer to be preempted. Within Ada the obvious way to do this is to execute "delay 0.0" (or delay until "yesterday"). This

technique is known as `deferred preemption' or `co-operative scheduling'. At least one major industrial user of Ada employs co-operative scheduling and is looking to see it standardized in the Standard. Note that non-preemptive behavior does not preclude interrupts either for the run-time system (to manage the delay queue) or for application-level interrupt handlers.

The definition of preemption in this AI follows the existing definition of FIFO_Within_Priority with the absence of the significant rule that a task switch must occur if a higher priority task is on a non-empty ready queue. Without this rule the current executing task will continue to execute unless it blocks, terminates or executes a delay statement.

The implementation permission allows the run-time system to ignore entries to and exits from protected objects. No priority changes need occur as preemption cannot occur during the time in which a task is executing within a normal (non-interrupt level) PO. This can lead to a very efficient implementation.

The above definitions have centered on the notion of a processor and non-preemptive execution on that processor. In Ada terms, however, this is not the complete story. Dispatching policies are set on a per-partition basis and it is possible for an implementation to put more than one partition on a processor. The Standard is silent about multi-partition scheduling, and there is clearly more than one way to schedule such a system, for example:

o use `priority' across all partitions on the same processor;

o assign overriding priorities to each partition;

o use priority dispatching within a partition and time-slicing between partitions.

The notion of `non-preemption' is different in all three cases. But since Ada only allows the dispatching policy within a partition to be defined, no further refinement of the language model can be given.

Full motivation and details of the scheduling analysis that can be applied to non-preemptive dispatching are contained in:
A. Burns, Defining New Non-Preemptive Dispatching and Locking Policies for Ada, Proceedings of Reliable Software Technologies - Ada Europe 2001, LNCS, pp328-336, 2001.

!example

!corrigendum D.2.4 (01)

@dinsc

A non-preemptive dispatching policy is defined by @i<policy_>@fa<identifier> Non_Preemptive_FIFO_Within_Priorities.

@i<@s8<Legality Rules>>

Non_Preemptive_FIFO_Within_Priorities can be specified as the @i<policy_>@fa<identifier> of pragma Task_Dispatching_Policy (see D.2.2).

@i<@s8<Post-Compilation Rules>>

If Non_Preemptive_FIFO_Within_Priorities is specified for a partition then Ceiling_Locking (see D.3) shall also be specified for that partition.

@i<@s8<Dynamic Semantics>>

When Non_Preemptive_FIFO_Within_Priorities is in effect, modifications to the ready queues occur only as follows:

@xbullet<When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.>

@xbullet<When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority.>

@xbullet<When a task executes a @fa<delay_statement> that does not result in blocking, it is added to the tail of the ready queue for its active priority. This is a task dispatching point (see D.2.1).>

@i<@s8<Implementation Permissions>>

Since implementations are allowed to round all ceiling priorities in subrange System.Priority to System.Priority'Last (see D.3), an implementation may allow a task to execute within a protected object without raising its active priority provided the protected object does not contain pragma Interrupt_Priority, Interrupt_Handler or Attach_Handler.

! ACATS Test

ACATS test(s) need to be created.

!appendix

!standard D.8
!class amendment

!status
!priority -- IRTAW11 consider this to be medium
!difficulty
!subject Non-Preemptive Dispatching
!from A. Burns on behalf of IRTAW11

! summary

A new dispatching policy and locking policy are defined for the non-preemptive execution of Ada tasks.

! problem

The Real-Time Annex requires that a preemptive dispatching policy is always provided by an implementation. It also allows implementation-defined policies to be supplied. For many system builders, particularly in the safety-critical area, the policy of choice is non-preemption.

The standard way of implementing many high-integrity applications is with a cyclic executive. Here a sequence of procedures is called within a defined time interval. Each procedure runs to completion, there is no concept of preemption. Data is passed from one procedure to another via shared variables; no synchronization constraints are needed since the procedures never run concurrently.

There is a need for a standard way of obtaining non-preemptive execution for Ada programs whilst still using tasks as a convenient means of encapsulating concurrent entities.

! proposal

Two new policy identifiers are defined, namely Non_Preemptive_FIFO_
Within_Priorities and Non_Preemptive_Locking. These are task dispatching policies and locking policies respectively and are used with the existing pragmas thus:

```
pragma Task_Dispatching_Policy (
        Non_Preemptive_FIFO_Within_Priorities);

pragma Locking_Policy (Non_Preemptive_Locking);
```

These policies are optional.

Post-Compilation Rules

If the Non_Preemptive_Locking policy is specified for a partition then Non_Preemptive_FIFO_Within_Priorities shall also be specified for that partition.

Dynamic Semantics

In accordance with implementation permission D.2.1 (9) an additional execution resource, the execution token, is defined.

Each processor has one such execution token.

A ready task must acquire the execution token before it can become the running task. When the Non_Preemptive_FIFO_Within_Priorities policy is in effect the modifications to the ready queues are identical to the existing preemptive policy FIFO_Within_Priorities.

The running task releases the execution token whenever it becomes suspended (or completed). It also releases the execution token whenever it executes a delay statement (whether this results in suspension or not). A new running task is selected and is assigned the execution token whenever the previously running task on that processor becomes suspended or otherwise releases the execution token. The rule for selecting the new running task follows the policy of Fifo_Within_Priorities.

On a multiprocessor system there may be further restrictions on where tasks may execute (as covered in D.2.1 (15)).

To cover asynchronous task interactions the following rules apply:

o If a task holding an execution token is aborted it releases the execution token when it completes.

o If a task holding an execution token executes a select-then-abort construct, and the trigger occurs, then the aborted construct is completed following the rules of D.6 but the token is retained.

o If a task holding an execution token is subject to a priority change (as a consequence of a call to Set_Priority, D.5) or asynchronous control (as a consequence of a call to Hold, D.11) then it retain the execution token. Note this can only occur in a multiprocessor implementation.

The locking policy, Non_Preemptive_Locking is defined as follows:

- if the protected object contains any of the following three pragmas:

Interrupt_Priority, Interrupt_Handler or Attach_Handler then the rules defined for locking policy Ceiling_Locking apply;

- if none of the above pragmas are present then, on a single processor, no run-time code need be generated to protect the object, in particular the priority of the calling task need not be changed;

- pragma Priority must not be present in any protected object (an alternative would be to say that pragma Priority has no effect).

NOTE:

* The running task may release the execution token by issuing a relative delay with a non-positive duration or an absolute delay using a time in the past, but be reassigned the token immediately if it is at the head of the highest priority ready queue.

* Implementation Permission 9.5.3 (22) still applies.

* It remains a bounded error to call a potentially blocking operation from within a protected object.

* A task executing an accept statement on which there is an outstanding call, proceeds without releasing the execution token (either before or after the execution of the accept statement). Select statements are treated similarly.

* A task calling an entry of a task releases the execution token even if the entry is open.

* It remains implementation defined, on a multiprocessor, whether a task waiting for access to a protected agent keeps the processor busy (i.e. retains the execution token), see D.2.1 (3).

! discussion

The above definitions have centered on the notion of a processor and non-preemptive execution on that processor. In Ada terms, however, this is not the complete story. Dispatching policies are set on a per-partition basis and it is possible for an implementation to put more than one partition on a processor. The standard is silent about multi-partition scheduling, and there is clearly more than one way to schedule such a system, for example:

o use `priority' across all partitions on the same processor;

o assign overriding priorities to each partition;

o use priority dispatching within a partition and time-slicing between partitions.

The notion of `non-preemption' is different in all three cases. But as Ada only allows the dispatching policy within a partition to be defined, no further refinement of the language model can be given. It would be possible for an implementation to have an execution token per processor or per partition.

! examples

! appendix

Full motivation and justification for the details of this proposal are contained in: A. Burns, Defining New Non-Preemptive Dispatching and Locking Policies for Ada, Proceedings of Reliable Software Technologies - Ada Europe 2001, LNCS, pp328-336, 2001.