Ada Issue 321 Definition of Dispatching Policies

!standard D.02   (01)   03-12-03 AI95-00321/05
!standard D.02.01 (01)
!standard D.02.01 (02)
!standard D.02.01 (04)
!standard D.02.01 (05)
!standard D.02.01 (06)
!standard D.02.01 (07)
!standard D.02.01 (08)
!standard D.02.01 (09)
!standard D.02.01 (10)
!standard D.02.02 (00)
!standard D.02.02 (03)
!standard D.02.02 (05)
!standard D.02.02 (07)
!standard D.02.02 (08)
!standard D.02.02 (09)
!standard D.02.02 (10)
!standard D.02.02 (11)
!standard D.02.02 (12)
!standard D.02.02 (13)
!standard D.02.02 (14)
!standard D.02.02 (15)
!standard D.02.02 (16)
!standard D.02.02 (17)
!standard D.02.02 (18)
!standard D.02.02 (19)
!standard D.02.02 (20)
!standard D.02.02 (21)
!standard D.02.03 (01)
!class amendment 03-01-07
!status Amendment 200Y 03-07-02
!status WG9 Approved 03-12-12
!status ARG Approved 13-0-0 03-06-23
!status work item 03-01-07
!status received 03-01-07
!priority Medium
!difficulty Medium
!subject Definition of dispatching policies

!summary

New wording is proposed for paragraphs within D.2.1 and D.2.2 to clarify the intended effect of dispatching points, and to allow more freedom to dispatching policies.

!problem

As it is currently worded the general model for dispatching (as defined in
D.2.1) does not permit a truly nonpreemptive dispatching policy. There are a number of required dispatching points that amount to preemption points, i.e., points that do not logically correspond to a decision by the running task to give up the processor.

During the discussion of this issue, it also appeared that the wording of D.2.1 does not make it clear enough that whenever a running task reaches a dispatching point it is conceptually added to one or more of the conceptual dispatching queues, and so is always eligible for consideration to continue execution.

!proposal

The solution is to restructure clauses D.2.1 and D.2.2 into a number of clauses: the basic task dispatching model; the pragma Task_Dispatching_Policy; and the specific policy FIFO_Within_Priorities. In so doing, we separate rules which are common to all task dispatching policies from rules for a specific policy.

For the detailed changes, see the !wording section.

!wording

D.2 Priority Scheduling

This clause describes the rules that determine which task is selected for execution when more than one task is ready (see 9).

D.2.1 The Task Dispatching Model

The task dispatching model specifies task scheduling, based on conceptual priority-ordered ready queues.

Dynamic Semantics

A task can become a running task only if it is ready (see 9) and the execution resources required by that task are available. Processors are allocated to tasks based on each task's active priority.

It is implementation defined whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy.

Task dispatching is the process by which one ready task is selected for execution on a processor. This selection is done at certain points during the execution of a task called task dispatching points. A task reaches a task dispatching point whenever it becomes blocked, and when it terminates.
Other task dispatching points are defined throughout this Annex for specific policies.

Task dispatching policies are specified in terms of conceptual ready queues and task states. A ready queue is an ordered list of ready tasks. The first position in a queue is called the head of the queue, and the last position is called the tail of the queue. A task is ready if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready queue of a processor contains exactly the set of tasks of that priority that are ready for execution on that processor, but are not running on any processor; that is, those tasks that are ready, are not running on any processor, and can be executed using that processor and other available resources. A task can be on the ready queues of more than one processor.

Each processor also has one running task, which is the task currently being executed by that processor. Whenever a task running on a processor reaches a task dispatching point it goes back to one or more ready queues; a task (possibly the same task) is then selected to run on that processor. The task selected is the one at the head of the highest priority nonempty ready queue; this task is then removed from all ready queues to which it belongs.

Implementation Permissions

An implementation is allowed to define additional resources as execution resources, and to define the corresponding allocation policies for them. Such resources may have an implementation-defined effect on task dispatching.

An implementation may place implementation-defined restrictions on tasks whose active priority is in the Interrupt_Priority range.

For optimization purposes, an implementation may alter the points at which task dispatching occurs, in an implementation-defined manner. However, a delay_statement always corresponds to at least one task dispatching point.

NOTES

7 Section 9 specifies under which circumstances a task becomes ready.The ready state is affected by the rules for task activation and termination, delay statements, and entry calls. When a task is not ready, it is said to be blocked.

8 An example of a possible implementation-defined execution resource is a page of physical memory, which needs to be loaded with a particular page of virtual memory before a task can continue execution.

9 The ready queues are purely conceptual; there is no requirement that such lists physically exist in an implementation.

10 While a task is running, it is not on any ready queue. Any time the task that is running on a processor is added to a ready queue, a new running task is selected for that processor.

11 In a multiprocessor system, a task can be on the ready queues of more than one processor. At the extreme, if several processors share the same set of ready tasks, the contents of their ready queues is identical, and so they can be viewed as sharing one ready queue, and can be implemented that way. Thus, the dispatching model covers multiprocessors where dispatching is implemented using a single ready queue, as well as those with separate dispatching domains.

12 The priority of a task is determined by rules specified in this subclause, and under D.1, ``Task Priorities'', D.3, ``Priority Ceiling Locking'', and D.5, ``Dynamic Priorities''.

13 The setting of a task's base priority as a result of a call to Set_Priority does not always take effect immediately when Set_Priority is called. The effect of setting the task's base priority is deferred while the affected task performs a protected action.

D.2.2 Pragma Task_Dispatching_Policy

Syntax

The form of a pragma Task_Dispatching_Policy is as follows:

pragma Task_Dispatching_Policy(policy_identifier);

Legality Rules

The policy_identifier shall either be one defined in this Annex or an implementation-defined identifier.

Post-Compilation Rules

A Task_Dispatching_Policy pragma is a configuration pragma.

Dynamic Semantics

A task dispatching policy specifies the details of task dispatching that are not covered by the basic task dispatching model. These rules govern when tasks are inserted into and deleted from the ready queues, and whether a task is inserted at the head or the tail of the queue for its active priority. The task dispatching policy is specified by a Task_Dispatching_Policy configuration pragma. If no such pragma appears in any of the program units comprising a partition, the task dispatching policy for that partition is unspecified.

Implementation Permissions

Implementations are allowed to define other task dispatching policies, but need not support more than one task dispatching policy per partition.

D.2.3 The Standard Task Dispatching Policy

This clause defines the policy_identifier, FIFO_Within_Priorities.

Post-Compilation Rules

If the FIFO_Within_Priorities policy is specified for a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition.

Dynamic Semantics

When 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, except in the case where the active priority is lowered due to the loss of inherited priority, in which case the task is added at the head of the ready queue for its new active priority.

o When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its 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.

Each of the events specified above is a task dispatching point (see D.2.1).

A task dispatching point occurs for the currently running task of a processor whenever there is a nonempty ready queue for that processor with a higher priority than the priority of the running task. The currently running task is said to be preempted and it is added at the head of the ready queue for its active priority.

Documentation Requirements

Priority inversion is the duration for which a task remains at the head of the highest priority nonempty ready queue while the processor executes a lower priority task. The implementation shall document:

o The maximum priority inversion a user task can experience due to activity of the implementation (on behalf of lower priority tasks), and

o whether execution of a task can be preempted by the implementation processing of delay expirations for lower priority tasks, and if so, for how long.

NOTES

14. If the active priority of a running task is lowered due to loss of inherited priority (as it is on completion of a protected operation) and there is a ready task of the same active priority that is not running, the running task continues to run (provided that there is no higher priority task).

15. Setting the base priority of a ready task causes the task to move to the tail of the queue for its active priority, regardless of whether the active priority of the task actually changes.

D.2.4 Non-Preemptive Dispatching

-- See AI-298.

D.2.5 ...

!discussion

This AI does not intend to make any change to the semantics of Ada 95 features.

The inclusion of 'completion of an accept_statement (see 9.5.2),' in D.2.1(4) was redundant for preemptive policies, and ruled out nonpreemptive policies. It is not the only case in which a task changes priority and/or unblocks another task. It has not been moved to D.2.3 as the overriding rule of higher priority task always preempting covers this and all similar cases.

Likewise, the inclusion of whenever a task 'becomes ready' was redundant. Dispatching points are defined only for running tasks. Becoming ready requires insertion into one or more ready queues, which invokes the preemption rule D.2.2(13) if the policy is preemptive and the newly ready task has higher priority than a running task for a processor to which the queue belongs.

!example

An example does not seem valuable for this proposal.

!corrigendum D.2(01)

@drepl
This clause describes the rules that determine which task is selected for execution when more than one task is ready (see 9.2). The rules have two parts: the task dispatching model (see D.2.1), and a specific task dispatching policy (see D.2.2).

@dby
This clause describes the rules that determine which task is selected for execution when more than one task is ready (see 9).

!corrigendum D.2.1(01)

@drepl
The task dispatching model specifies preemptive scheduling, based on conceptual priority-ordered ready queues.
@dby
The task dispatching model specifies task scheduling, based on conceptual priority-ordered ready queues.

!corrigendum D.2.1(02)

@drepl
A task runs (that is, it becomes a @i<running task>) only when it is ready (see
9.2) and the execution resources required by that task are available.
Processors are allocated to tasks based on each task's active priority.
@dby
A task can become a @i<running task> only if it is ready (see 9) and the execution resources required by that task are available. Processors are allocated to tasks based on each task's active priority.

!corrigendum D.2.1(04)

@drepl
@i<Task dispatching> is the process by which one ready task is selected for execution on a processor. This selection is done at certain points during the execution of a task called @i<task dispatching points>. A task reaches a task dispatching point whenever it becomes blocked, and whenever it becomes ready.
In addition, the completion of an @fa<accept_statement> (see 9.5.2), and task termination are task dispatching points for the executing task. Other task dispatching points are defined throughout this Annex.
@dby
@i<Task dispatching> is the process by which one ready task is selected for execution on a processor. This selection is done at certain points during the execution of a task called @i<task dispatching points>. A task reaches a task dispatching point whenever it becomes blocked, and when it terminates.
Other task dispatching points are defined throughout this Annex for specific policies.

!corrigendum D.2.1(05)

@drepl
@i<Task dispatching policies> are specified in terms of conceptual @i<ready queues>, task states, and task preemption. A ready queue is an ordered list of ready tasks. The first position in a queue is called the @i<head of the queue>, and the last position is called the @i<tail of the queue>. A task is @i<ready> if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready queue of a processor contains exactly the set of tasks of that priority that are ready for execution on that processor, but are not running on any processor; that is, those tasks that are ready, are not running on any processor, and can be executed using that processor and other available resources. A task can be on the ready queues of more than one processor.
@dby
@i<Task dispatching policies> are specified in terms of conceptual @i<ready queues> and task states. A ready queue is an ordered list of ready tasks. The first position in a queue is called the @i<head of the queue>, and the last position is called the @i<tail of the queue>. A task is @i<ready> if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready queue of a processor contains exactly the set of tasks of that priority that are ready for execution on that processor, but are not running on any processor; that is, those tasks that are ready, are not running on any processor, and can be executed using that processor and other available resources. A task can be on the ready queues of more than one processor.

!corrigendum D.2.1(06)

@drepl
Each processor also has one @i<running task>, which is the task currently being executed by that processor. Whenever a task running on a processor reaches a task dispatching point, one task is selected to run on that processor. The task selected is the one at the head of the highest priority nonempty ready queue; this task is then removed from all ready queues to which it belongs.
@dby
Each processor also has one @i<running task>, which is the task currently being executed by that processor. Whenever a task running on a processor reaches a task dispatching point it goes back to one or more ready queues; a task (possibly the same task) is then selected to run on that processor. The task selected is the one at the head of the highest priority nonempty ready queue; this task is then removed from all ready queues to which it belongs.

!corrigendum D.2.1(07)

@ddel
A preemptible resource is a resource that while allocated to one task can be allocated (temporarily) to another instead. Processors are preemptible resources. Access to a protected object (see 9.5.1) is a nonpreemptible resource. {preempted task} When a higher-priority task is dispatched to the processor, and the previously running task is placed on the appropriate ready queue, the latter task is said to be @i<preempted>.

!corrigendum D.2.1(08)

@ddel
A new running task is also selected whenever there is a nonempty ready queue with a higher priority than the priority of the running task, or when the task dispatching policy requires a running task to go back to a ready queue. These are also task dispatching points.

!corrigendum D.2.1(09)

@drepl
An implementation is allowed to define additional resources as execution resources, and to define the corresponding allocation policies for them. Such resources may have an implementation defined effect on task dispatching (see D.2.2).
@dby
An implementation is allowed to define additional resources as execution resources, and to define the corresponding allocation policies for them. Such resources may have an implementation-defined effect on task dispatching.

!corrigendum D.2.1(10)

@dinsa
An implementation may place implementation-defined restrictions on tasks whose active priority is in the Interrupt_Priority range.
@dinst
For optimization purposes, an implementation may alter the points at which task dispatching occurs, in an implementation-defined manner. However, a @fa<delay_statement> always corresponds to at least one task dispatching point.

!corrigendum D.2.1(16)

@dinsa

@xindent<@s9<12 The priority of a task is determined by rules specified in this subclause, and under D.1, ``Task Priorities'', D.3, `` Priority Ceiling Locking'', and D.5, ``Dynamic Priorities''.>>
@dinst
@xindent<@s9<13 The setting of a task's base priority as a result of a call to Set_Priority does not always take effect immediately when Set_Priority is called. The effect of setting the task's base priority is deferred while the affected task performs a protected action.>>

!corrigendum D.02.02(00)

@drepl
The Standard Task Dispatching Policy
@dby
Pragma Task_Dispatching_Policy

!corrigendum D.02.02(03)

@drepl
The @i<policy>_@fa<identifier> shall either be FIFO_Within_Priorities or an implementation-defined @fa<identifier>.
@dby
The @i<policy>_@fa<identifier> shall either be one defined in this Annex or an implementation-defined @fa<identifier>.

!corrigendum D.02.02(05)

@ddel
If the FIFO_Within_Priorities policy is specified for a partition, then the
Ceiling_Locking policy (see D.3) shall also be specified for the partition.

!corrigendum D.02.02(07)

@ddel
The language defines only one task dispatching policy, FIFO_Within_Priorities; when this policy is in effect, modifications to the ready queues occur only as follows:

!corrigendum D.02.02(08)

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

!corrigendum D.02.02(09)

@ddel
@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, except in the case where the active priority is lowered due to the loss of inherited priority, in which case the task is added at the head of the ready queue for its new active priority.>

!corrigendum D.02.02(10)

@ddel

@xbullet<When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its active priority.>

!corrigendum D.02.02(11)

@ddel
@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.>

!corrigendum D.02.02(12)

@ddel
Each of the events specified above is a task dispatching point (see D.2.1).

!corrigendum D.02.02(13)

@ddel
In addition, when a task is preempted, it is added at the head of the ready queue for its active priority.

!corrigendum D.02.02(14)

@ddel
@i<Priority inversion> is the duration for which a task remains at the head of the highest priority ready queue while the processor executes a lower priority task. The implementation shall document:

!corrigendum D.02.02(15)

@ddel
@xbullet<The maximum priority inversion a user task can experience due to activity of the implementation (on behalf of lower priority tasks), and>

!corrigendum D.02.02(16)

@ddel
@xbullet<whether execution of a task can be preempted by the implementation processing of delay expirations for lower priority tasks, and if so, for how long.>

!corrigendum D.02.02(17)

Implementations are allowed to define other task dispatching policies, but need not support more than one such policy per partition.
@dby
Implementations are allowed to define other task dispatching policies, but need not support more than one task dispatching policy per partition.

!corrigendum D.02.02(18)

@ddel
For optimization purposes, an implementation may alter the points at which task dispatching occurs, in an implementation defined manner. However, a
@i<delay_statement> always corresponds to at least one task dispatching point.

!corrigendum D.02.02(19)

@ddel

@xindent<@s9<13 If the active priority of a running task is lowered due to loss of inherited priority (as it is on completion of a protected operation) and there is a ready task of the same active priority that is not running, the running task continues to run (provided that there is no higher priority task).>>

!corrigendum D.02.02(20)

@ddel
@xindent<@s9<14 The setting of a task's base priority as a result of a call to Set_Priority does not always take effect immediately when Set_Priority is called. The effect of setting the task's base priority is deferred while the affected task performs a protected action.>>

!corrigendum D.02.02(21)

@ddel
@xindent<@s9<15 Setting the base priority of a ready task causes the task to move to the end of the queue for its active priority, regardless of whether the active priority of the task actually changes.>>

!corrigendum D.02.03(01)

@dinsc

This clause defines the policy_identifier, FIFO_Within_Priorities.

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

If the FIFO_Within_Priorities policy is specified for a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition.

@i<@s8<Dynamic Semantics>>

When 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, except in the case where the active priority is lowered due to the loss of inherited priority, in which case the task is added at the head of the ready queue for its new active priority.>

@xbullet<When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its 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.>

Each of the events specified above is a task dispatching point (see D.2.1).

A task dispatching point occurs for the currently running task of a processor whenever there is a nonempty ready queue for that processor with a higher priority than the priority of the running task. The currently running task is said to be @i<preempted> and it is added at the head of the ready queue for its active priority.

@i<@s8<Documentation Requirements>>

@i<Priority inversion> is the duration for which a task remains at the head of the highest priority nonempty ready queue while the processor executes a lower priority task. The implementation shall document:

@xbullet<The maximum priority inversion a user task can experience due to activity of the implementation (on behalf of lower priority tasks), and>

@xbullet<whether execution of a task can be preempted by the implementation processing of delay expirations for lower priority tasks, and if so, for how long.>

@xindent<@s9<NOTES@hr
14 If the active priority of a running task is lowered due to loss of inherited priority (as it is on completion of a protected operation) and there is a ready task of the same active priority that is not running, the running task continues to run (provided that there is no higher priority task).>>

@xindent<@s9<15 Setting the base priority of a ready task causes the task to move to the tail of the queue for its active priority, regardless of whether the active priority of the task actually changes.>>

!ACATS test

Since this AI is rearranging the text of the standard, but not (intentionally) changing the semantics, no test is needed.