Real-time CORBA 2.0: Dynamic Scheduling Specification

This OMG document replaces the draft adopted specification and submission (orbos/01-06-09). It is an OMG Final Adopted Specification, which has been approved by the OMG board and technical plenaries, and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by March 1, 2002.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*; however, at the time of this writing there were no pending issues.

The FTF Recommendation and Report for this specification will be published on April 1, 2002. If you are reading this after that date, please download the available specification from the OMG formal specifications web page.

OMG Final Adopted Specification
September 2001

Real-Time CORBA 2.0: Dynamic Scheduling 26

Note – Eventually, this chapter will become part of CORBA Core. The chapter number is temporary.

Contents

This chapter contains the following topics.

Section I - Overview and Rationale

26.1 Overview

26.1.1 Dynamic Scheduling

In real-time, we distinguish between two types of distributed systems based on how the system is used, and its impact on the underlying infrastructure. There are *static* and *dynamic* distributed systems.

Static distributed systems are those where the processing load on the system is within known bounds such that *a priori* analysis can be performed. This means that the set of applications that the system could be running is know in advance, and that the workload that will be imposed on each application can be predicted within a known bound. Such systems often have a limited number of application configurations that can be executed, sometimes referred to as system modes. In these systems, a schedule for the execution of applications can be worked out in advance for each system mode, with a bounded amount of variation. As a result, the underlying infrastructure (operating system and middle-ware) need only be able to support executing that schedule.

One common approach to static systems is the use of Operating System Priorities to manage deadlines. Offline analysis is performed to map different application temporal requirements (such as frequency of execution) onto the available priorities. If the underlying infrastructure always respects these priorities, including preempting low priority threads when higher priority threads become eligible to run, and providing priority inheritance, this is sufficient.

Dynamic distributed systems, on the other hand, do not have a sufficiently predictable workload to allow this approach. It may be that the set of applications is either too large or not known in advance, the processing requirements for an application is too variable to be pre-planned, the arrival time of the inputs is too variable, or some other source of variability. For these types of systems, the underlying infrastructure must be able to satisfy real-time requirements in a dynamically changing environment.

This specification is focused on systems in which the discipline for scheduling CORBA ORB and application threads (e.g., highest priority first, or earliest deadline first, or least laxity first) may be chosen by the application or system designers; and the scheduling input values needed by a scheduler to control execution (called *scheduling parameter elements* in this document) for that scheduling discipline (e.g., priority, deadline, expected execution time) may be changed by the application dynamically (i.e., at any time). In contrast, Real-time CORBA 1.0 (ORBOS/99-02-12 with ORBOS/99-03-29) is focused on fixed priority systems.

26.1.2 Distributable Thread

This specification replaces the term and concept of an *activity* that appeared as a design and analysis suggestion in Real-time CORBA 1.0 with a specification for an end-to-end schedulable entity termed *distributable thread*. Additionally, this specifications introduction of the entity termed *scheduling segment* completes the replacement of the *activity* concept from Real-time CORBA 1.0.

26.2 Rationale

Dynamic scheduling is widely employed in real-time and distributed real-time computing systems. This specification extends Real-time CORBA 1.0 to encompass these dynamic systems as well as static systems.

In most real-time systems, especially distributed systems, cost-effectiveness demands that the computing system employ as much application-specific knowledge about the application and its execution environment as feasible. Much of this knowledge can be best captured in the scheduling discipline. This specification allows such application-specific scheduling disciplines to be implemented by a pluggable scheduler.

An end-to-end execution model is essential to achieving end-to-end predictability of timeliness in a distributed real-time computing system. This is especially important in dynamically scheduled systems. The end-to-end execution model may be provided according to a formal standard specification (as herein), or as an ad hoc, custom-made creation by multiple different application programmers.

26.3 Notional Scheduling Service Architecture

This specification is based on a scheduling service architecture for a hypothetical or notional scheduling service plug-in. The specification does not require that scheduling service implementations conform to this notional architecture. It is presented only to provide an aid to understanding and describing the specification of a generic scheduling service framework.

The application interacts with the scheduler, passing information about its (the application's) scheduling needs and its predicted use of system resources. The scheduler is responsible for determining how best to meet the schedule given that resource usage. The scheduler will use one or more scheduling disciplines, such as Earliest Deadline First or Maximum Urgency First, to achieve this goal.

The application must also interact with the scheduler whenever there is a significant change in its scheduling needs or its predicted resource usage. In addition, the application must ensure that the scheduler is able to run as often as it needs to in order to maintain the schedule – the application should not, for example, disable interrupts or pre-emption for long periods of time, or create high priority threads that the scheduler does not know about. If there are insufficient naturally occurring interaction points, the application must include some additional interactions with the scheduler just to guarantee that overruns and other errors can be detected in a timely way.

In a CORBA environment, the application can take an action (for example, making a CORBA request) that could impact the schedule. In the notional architecture, the ORB is responsible for interacting with the scheduler at these points, so that the scheduler can take into account the transitioning of control from one processing node to another. Thus, a set of specific ORB-scheduler interfaces are defined.

26.4 Goals of this Specification

This specification generalizes the Real-time CORBA 1.0 (ORBOS/99-02-12 and ORBOS/99-03-29) specification to meet the requirements of a much greater segment of the real-time computing field. There are three major generalizations:

- any scheduling discipline may be employed;
- the scheduling parameter elements associated with the chosen discipline may be changed at any time during execution;
- the schedulable entity is a *distributable thread* that may span node boundaries, carrying its scheduling context among scheduler instances on those nodes.

While the Real-time CORBA 1.0 Scheduling Service interfaces have been replaced, this specification is backward compatible with the semantics of the Scheduling Service defined in the Real-time CORBA 1.0 specification. Compatible implementations of both specifications may be used in different ORB instances within the same system. Not all features of this specification can be used in such mixed systems.

This specification imposes no requirements on base real-time operating systems, other than the conventional ability to dispatch threads in a pre-emptive fashion. This specification imposes no additional constraints on the real-time operating system beyond those in Real-time CORBA 1.0.

26.5 Scope

This specification adds interfaces for a small set of well known scheduling disciplines to CORBA as optional compliance points. This specification does not attempt to provide all the interfaces necessary for interoperability of dynamically scheduled

applications and schedulers in heterogeneous systems. Rather, this specification provides a framework upon which schedulers can be built and lays the foundation for future full interoperability, and provides sufficient interfaces for applications to be built using the set of included scheduler disciplines.

This specification defines a set of ORB/scheduler interfaces that will allow the development of portable (i.e., ORB implementation independent) schedulers. For the defined disciplines, this specification also specifies interfaces that will allow the development of portable (i.e. ,ORB and scheduler implementation independent) applications. The specification of portable application interfaces for other scheduling disciplines is left to future revisions.

Note that most scheduler implementations will extensively utilize features of the underlying operating system, and in some cases the networking software. This aspect of scheduler implementation is outside of the scope of this specification. Therefore, this specification does not provide the portability of schedulers except with respect to ORB interactions.

This specification does not provide interoperability between scheduling disciplines and thus not between different scheduler implementations. A scheduling framework is provided and the mechanism used for passing information between scheduler instances is provided via GIOP service contexts. However, the format and content of the information passed in the GIOP service contexts are not specified. On-the-wire interoperability between scheduling disciplines and the corresponding scheduler implementations is left to future specifications.

This specification provides an abstraction for distributed real-time programming (the *distributable thread*). This specification does not attempt to address more advanced issues such as fault tolerance, propagation of system information and control along the path of a distributable thread, etc. These facilities may be provided in a subsequent revision of this specification.

Section II - Concepts

26.6 Sequencing: Scheduling and Dispatching

Usually multiple execution entities (hereafter referred to as “threads”) contend for one or more exclusively accessed resources – notably processor cycles, but also others, both physical (e.g., communication paths) and logical (e.g., synchronizers). This contention must be resolved into a sequence of resource accesses – e.g., thread executions. In general, contention for all shared resources should be resolved in a consistent manner, although this is not yet common practice – e.g., processors may be allocated by priority, networks by first come first served, locks by serializability, disks by head movement distance, etc. All resource contention can be resolved by one of two sequencing means: either scheduling or dispatching.

Thread *scheduling* is deciding in what order they all will execute. Each time thread scheduling is performed, a sequence is established – a schedule – for all threads ready at that time. Scheduling is performed *statically* (prior to execution time), by a person or a program, or *dynamically* (at execution time) by a user or the system software.

Thread *dispatching* is granting resource access – e.g., running the currently most eligible thread. When scheduling is employed, dispatching occurs in schedule order.

Thread scheduling is not always necessary nor computationally feasible – dispatching alone may be sufficient.

Moreover, some actions are never threads and thus not schedulable – most commonly, interrupt service routines and certain OS services, which execute either when invoked or automatically as needed (other OS services are scheduled in concert with application entities).

Dispatching, when scheduling is not employed, establishes a thread resource access – e.g., execution sequence – one thread or non-schedulable action at a time.

The execution sequence may change at a sequencing point (either a *scheduling point* or a *dispatching point*), such as when a thread becomes ready or blocked, or a thread contends for a resource, or a thread time constraint is violated.

Contention for execution (and all other sequentially shared physical and logical resources) generally should be resolved according to an application-specific *sequencing optimality criterion* that seeks maximal usefulness to the system. In real-time systems, that usefulness is based primarily on (but not limited to) timeliness and predictability of timeliness. (Other factors, not related to real-time, commonly found in sequencing criteria include relative importance, precedence constraints, resource ownership, etc.)

Sequencing optimality criteria are what define *timeliness* for a given system or application. Consequently, they also distinguish hard and soft real-time. *Hard real-time* has a single timeliness factor in its sequencing optimality criterion: always meet all hard deadlines. *Soft real-time* includes all other possible timeliness factors in sequencing (usually scheduling) optimality criteria – very common examples are “minimize mean weighted tardiness,” “minimize the number of missed deadlines according to importance,” and “minimize maximum tardiness.”

Informally, a property is *predictable* to the degree that it is known in advance. One end point of the predictability scale is *determinism*, in the sense that the property is known exactly in advance. The other end point of the predictability scale can be characterized as maximum entropy, in the sense that nothing at all is known in advance about the property. In stochastic real-time systems (which include hard real-time systems as a special case), one well-defined way to measure predictability is coefficient of variation C_v, which is defined as variance/mean^2. The deterministic distribution, $C_v = 0$, and the extreme mixture of exponentials distribution is an example of a maximally non-deterministic property whose $C_v = \infty$.

In every real-time system, timeliness of each application and system action is somewhere on this predictability scale. Hard real-time systems have deterministic timeliness in the sense that they always meet all of their hard deadlines. Soft real-time systems have non-deterministic timeliness – e.g., characterized stochastically, such as minimizing either mean or maximum tardiness.

Given a sequencing optimality criterion, a *sequencing discipline* is selected or devised to satisfy it. There are a great many widely used sequencing disciplines; common examples in real-time computing systems include highest priority first (or just "priority"), earliest deadline first (EDF), and least laxity first (LLF). There may be more than one discipline that satisfies a given criterion – e.g., the hard real-time criterion is satisfied by: the EDF and LLF disciplines (under specific conditions), among others; or appropriate assignment and manipulation of priorities. Conversely, a specific discipline may be suitable for different criteria: EDF satisfies the hard real-time criterion, and also satisfies the soft real-time criterion "minimize maximum tardiness" (among others); priorities can be used to satisfy either the hard or various soft real-time criteria.

When scheduling is employed, the sequencing discipline is usually called a *scheduling discipline*, and when only dispatching is employed, the discipline is usually called a *dispatching rule*.

A *sequencing algorithm* implements a sequencing discipline. In general, a discipline can be implemented by many different possible algorithms.

This specification uses the term *scheduling* to include the case when scheduling (and thus dispatching in schedule order) is employed, and the case when only dispatching is employed, because both of those cases involve selecting a sequencing optimality criterion and a corresponding discipline and algorithm.

26.7 Well Known Scheduling Disciplines

There are many widely used scheduling disciplines, but real-time computing theory and practice are focused on a small number of them, some of which are summarized below. The constructs in the IDL will become clear later in this specification.

26.7.1 Fixed Priority Scheduling

The fixed priority scheduling discipline provides for pre-emptive scheduling or dispatching of threads based on a simple numeric priority. When a higher priority thread is created or becomes unblocked, it pre-empts a lower priority executing thread and executes immediately.

```
module FP_Scheduling
{
    struct SegmentSchedulingParameter
    {
        RTCORBA::Priority base_priority;
    };
```

```
    local interface SegmentSchedulingParameterPolicy
        : CORBA::Policy
    {
        attribute SegmentSchedulingParameter value;
    };

    struct ResourceSchedulingParameter
    {
        RTCORBA::Priority resource_priority_ceiling;
    };

    local interface ResourceSchedulingParameterPolicy
        : CORBA::Policy
    {
        attribute ResourceSchedulingParameter value;
    };

    local interface Scheduler
        : RTScheduling::Scheduler
    {
        SegmentSchedulingParameterPolicy
            create_segment_scheduling_parameter
                (in SegmentSchedulingParameter value);

        ResourceSchedulingParameterPolicy
            create_resource_scheduling_parameter
                (in ResourceSchedulingParameter value);
    };
};
```

Note that an analysis technique for scheduling fixed priority systems is Rate Monotonic Analysis (RMA). The rate monotonic analysis assigns fixed priorities to periodic threads based on their execution rates or periods – the thread having the highest rate (shortest period) is assigned the highest priority. Normally, the characteristics of all threads and their execution environment are known in advance, and rate monotonic scheduling is statically performed off-line. In this case, the Real-time CORBA 1.0 Fixed Priority discipline can be employed. Often thread behavior or execution environment characteristics such as system loading vary with some dynamic parameter, time or date, operational status of supporting systems, etc.

26.7.2 Earliest Deadline First (EDF)

The earliest deadline first discipline uses the execution completion deadline of the threads as the basis for their execution eligibility – a thread that has a shorter (closer) deadline is more eligible than one with a longer (later) deadline. In some cases, a thread's deadline is constant during the thread's lifetime, and in other cases it changes (for example, a thread's deadlines may be nested). When EDF is used to meet deadlines (i.e., for hard real-time), it requires that all deadlines can be met, in which case it is most often employed statically. Other factors can be used in conjunction with deadlines to create enhanced EDF-like disciplines that always meet all deadlines if

possible, and that shed or defer load when overloaded. When EDF is used to minimize maximum tardiness (i.e., for soft real-time), it may be employed either statically or dynamically. EDF can be employed either as a scheduling discipline or as a dispatching rule.

```
module EDF_Scheduling
{

    struct SchedulingParameter
    {
        TimeBase::TimeT deadline;
        long            importance;
    };

    local interface SchedulingParameterPolicy
        : CORBA::Policy
    {
        attribute SchedulingParameter value;
    };

    local interface Scheduler : RTScheduling::Scheduler
    {
        SchedulingParameterPolicy
            create_scheduling_parameter
                (in SchedulingParameter value);
    };
};
```

26.7.3 Least Laxity First (LLF)

A least laxity (or “time to go”) first discipline assigns execution eligibility based on laxity value, where

laxity = deadline - current time - estimated remaining computation time.

A thread with lower laxity is more eligible than one with higher laxity. An LLF discipline is sometimes used for environments where thread execution time requirements vary significantly. In such environments, a thread with a long execution time may be released prior to threads with less laxity becoming ready-to-run. The laxity estimate is updated as the thread execution duration estimate is updated at run time. An LLF discipline may specify that a thread with negative laxity should not (continue to) execute. Thus, LLF is primarily a dynamic discipline. LLF may be used either to meet deadlines (i.e., for hard real-time) or to maximize minimum lateness (or tardiness) (i.e., for soft real-time). LLF can be employed either as a scheduling discipline or a dispatching rule.

```
module LLF_Scheduling
{
    struct SchedulingParameter
    {
```

```
        TimeBase::TimeT deadline;
        TimeBase::TimeT estimated_initial_execution_time;
        long        importance;
    };
    // laxity = deadline
    //        - {current time}
    //        - (estimated_initial_execution_time –
    //                  {time executed thus far})

    local interface SchedulingParameterPolicy
        : CORBA::Policy
    {
        attribute SchedulingParameter value;
    };

    local interface Scheduler : RTScheduling::Scheduler
    {
        SchedulingParameterPolicy
            create_scheduling_parameter
                (in SchedulingParameter value);
    };
};
```

26.7.4 Maximize Accrued Utility (MAU)

A maximize accrued utility discipline uses a utility function associated with each thread to establish a thread schedule; MAU cannot be used as a dispatching rule. Each function provides a mapping from that thread's completion time to a utility value. For example, completing very close to but prior to the deadline may be most useful, while completing much earlier than the deadline may have less utility, and completing after the deadline may have zero or negative utility. Conventional deadlines are a special case of utility functions. MAU disciplines seek schedules that result in maximal accrued (e.g., summed) utility. Thus, MAU disciplines are intended for dynamic systems.

```
module Max_Utility_Scheduling
{
    struct SchedulingParameter
    {
        TimeBase::TimeT deadline;
        long        importance;
    };

    local interface SchedulingParameterPolicy
        : CORBA::Policy
    {
        attribute SchedulingParameter value;
    };

    local interface Scheduler : RTScheduling::Scheduler
```

```
    {
        SchedulingParameterPolicy
            create_scheduling_parameter
                (in SchedulingParameter value);
    };
};
```

26.8 Distributed System Scheduling

Scheduling in a distributed system can be divided into four cases. Cases 1 through 3 are the single-level scheduling cases.

Case 1 is that scheduling occurs completely independently on each node, and application behaviors that involve more than a single node do not have trans-node end-to-end timeliness requirements that are used by the node schedulers. That is the common non-real-time case.

Case 2 is that scheduling occurs independently on each node, but an application behavior that involves more than one node propagates its end-to-end timeliness context, which is then used by each node's scheduler while the behavior is active at that node. At each node, the distributable thread competes with the other threads at that node, according to its end-to-end needs. System-wide scheduling is coherent but not generally globally optimal. That is the distributed real-time case this specification explicitly addresses.

Case 3 is that scheduling on each node is global in the sense that there is a logically singular system-wide scheduling algorithm instantiated on all nodes, and node instances of this algorithm interact to cooperatively schedule all nodes in a globally optimal way. This specification does not explicitly support case 3 because such scheduling is very difficult (intractable in general) from both the conceptual and implementation standpoints, but desires not to preclude it.

Case 4 is all the multi-level scheduling cases: there is at least one level of "meta-scheduling" above the case 1 or 2 node schedulers that seeks to improve global optimality by adaptively adjusting some combination of scheduling parameter elements, schedulable entity, scheduling contexts, scheduling algorithms, scheduling disciplines, and node load balancing. Case 4 includes sub-cases corresponding to cases 2 and 3. This specification does not explicitly support case 4, but again, desires not to preclude it.

26.9 Distributable Thread

This specification replaces the term and concept of an *activity* that appeared as a design and analysis suggestion in Real-time CORBA 1.0 with a definition for an end-to-end schedulable entity termed *distributable thread*. Real-time CORBA 1.0 left the details of the activity abstraction unspecified, but such a distributed execution model is essential in dynamically scheduled real-time CORBA systems. The term has been changed to avoid conflict with prior usage in CORBA specifications such as Workflow Management (formal/00-05-02) and Additional Structuring Mechanisms for the OTS Specification (orbos/00-04-02).

The defining characteristic of any real-time distributed computing system, whatever its programming model, is that the end-to-end timeliness (optimality and predictability of optimality, as defined in Section 26.6, “Sequencing: Scheduling and Dispatching,” on page 26-5) of trans-node application behaviors is acceptable to the application.

In most cases, the fundamental requirement for achieving acceptable end-to-end timeliness is that a trans-node application behavior’s timeliness properties and parameters – time constraints, expected execution time, execution time received thus far, etc. – be explicitly employed for resource management (scheduling, etc.) consistently on each node involved in that application trans-node behavior. As stated in Section 26.8, “Distributed System Scheduling,” on page 26-11, “consistently” refers to distributed scheduling case 3 in this specification.

In some static real-time distributed systems, these properties and parameters can be instantiated *a priori*; but in general, and in all dynamic real-time distributed systems, these properties must be propagated among corresponding computing node resource managers – e.g., in operating systems, Java virtual machines, and middleware.

In real-time distributed CORBA systems, a trans-node application behavior’s end-to-end timeliness (and potentially other) properties and parameters must be acquired from the client’s ORB, propagated with the invocation, and deposited in the servant’s ORB, when operation invocations occur; and similarly for any associated returns.

For (at least) dynamic real-time CORBA systems, the fixed priority propagation mechanism in the Real-time CORBA 1.0 specification is insufficient – a trans-node application behavior abstraction is needed. A natural abstraction is suggested by CORBA’s native control flow programming model – a thread that can execute operations in objects without regard for physical node boundaries. In this specification, that programming model abstraction is termed a *distributable thread* (see Figure 26-1). In this figure, Objects A, B, and C may be running on either different processing nodes or within the same address space.

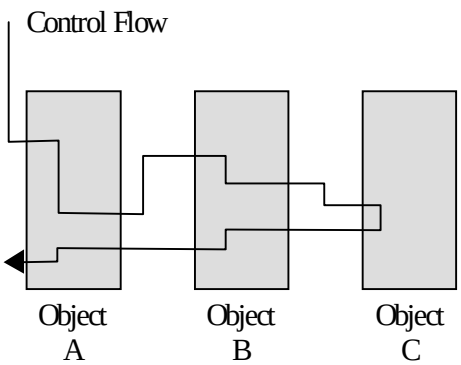

Figure 26-1 Control Flow in a Distributed Processing System

The distributable thread is the schedulable entity. Each distributable thread has a unique system-wide id. Each distributable thread may have one or more execution scheduling parameter elements – e.g., priority, time constraints such as deadlines or utility functions, importance – that specify the acceptable end-to-end timeliness for completing the sequential execution of operations in object instances that may reside on multiple physical nodes. The semantics of acceptability with respect to these end-to-end timeliness parameters is defined by the application, in the context of the scheduling discipline being used. Execution of the distributable thread is governed by the scheduling parameter elements, on each node it visits (see Figure 26-2).

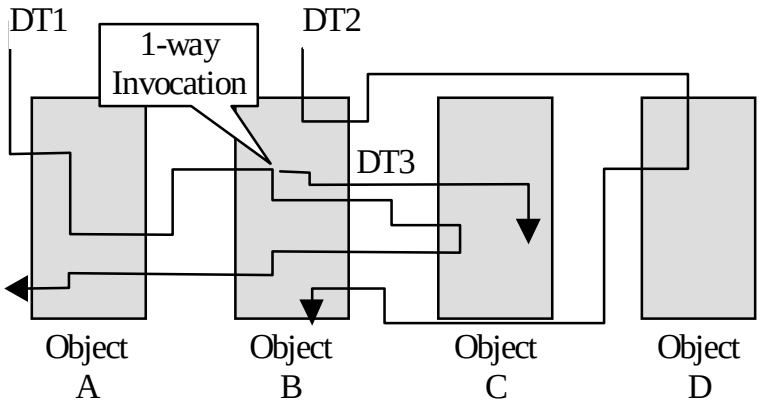

Figure 26-2 Distributed Threads

A distributable thread can extend and retract its locus of execution points among operations in object instances across physical computing nodes by location-independent invocations and (optionally) returns. Within each node, the flow of control is equivalent to normal local thread execution.

The synchrony of a conventional two-way operation invocation (or RPC) programming model is often cited as a concurrency limitation. But that criticism does not apply to the distributable thread model. A distributable thread is a sequential abstraction, like a local thread. A distributable thread is always executing somewhere, while it is the most eligible there – it is not doing send/wait's as with conventional operation invocations.

Remote invocations and returns are scheduling events at both client and servant nodes. Each node's processor is always executing the most eligible distributable thread while the others wait.

A distributable thread always has exactly one execution point (*head*) in the whole system. New distributable threads may be created, or sleeping ones awakened, when needed. An application or system may have multiple distributable threads. Multiple distributable threads execute concurrently and asynchronously, by default. Distributable threads synchronize through operation execution; the writers of each object control distributable thread concurrency in that object.

An exception that occurs anywhere along a distributable thread's locus of execution can be forwarded to and raised at the head of that distributable thread. Subsequently, the exception propagates from the head back up the distributable thread to the nearest enclosing exception handler.

Distributable thread-based programming models imply the need for a number of supporting facilities; these programming models can be differentiated by the facilities that they provide, and the approaches employed to provide them. Not all, or any, of these facilities are included in this specification. These supporting facilities include (but are not limited to) the following, which are not addressed in this specification:

- Some asynchronous "happenings" (i.e., changes in system state) of interest to a distributable thread may have to be coordinated with current distributable thread execution. For example, a violated time constraint, or the failure of a node or network path over which a distributable thread is extended, might require notification of the distributable thread's head – as soon as possible, if the distributable thread is currently executing, and otherwise as soon as the distributable thread becomes the most eligible to execute.

  Certain other events that occur at the distributable thread's head – e.g., synchronous exceptions (e.g., traps) and asynchronous exceptions (e.g., time constraint expirations) – may require the distributable thread to execute a local exception handler and then return back up the invocation chain to execute one or more appropriate exception handlers at those places. After such an exception, the programming model could allow the distributable thread to either continue execution where the exception was initially delivered (a *continuation* model) or terminate, or the model could require that the distributable thread always terminate (a *termination* model).

- Distributable thread control actions – e.g., suspend, resume, abort, time constraint change, etc. – may have to be propagated to, and carried out at, the distributed thread's head.

- Mechanisms may have to be provided to support maintaining correctness of distributed execution, and consistency of distributed data – in both cases, as defined by the application – for concurrent activities of one or more applications.

- The code that is responsible for detecting/suspecting failure for an appropriate set of nodes may require visibility to failures locally perceived by a distributable thread.

All of these facilities generally would be required to be timely – e.g., subject to completion time constraints.

Section III - Overview of the Programming Model

This section presents an overview of the application programming model that is being provided. Since the specification defines a scheduling framework, as well as a limited set of scheduling disciplines, this section deals with the concepts that apply across schedulers and scheduling disciplines.

26.10 Scheduler

In this specification a scheduler is realized as a extension to Real-time CORBA that utilizes the scheduling needs and resource requirements of one or more applications to manage the order of execution of those applications on the distributed nodes of a CORBA system. A scheduler provides operations for applications to announce their requirements, which the scheduler takes into consideration when it affects the order in which threads are dispatched by the operating system.

A scheduler will be run in response to specific application requests, such as defining new scheduling parameter elements, and in response to specific application actions, such as CORBA invocations. The latter will be implemented using the CORBA Portable Interceptor interfaces. The scheduler utilizes the information provided in these interfaces to manipulate which threads are most eligible for execution by the underlying operating system. This control is via whatever interfaces the operating system provides, which are outside of the scope of CORBA. Thus, although scheduler implementations could be independent of any particular ORB implementation, as long as the ORB conforms to this specification, the scheduler will be closely tied to the operating system.

The scheduler architecture is based on the premise that a distributed application can be considered to be a set of distributable threads (see Section Distributable Thread), which may interact in a number of ways, sharing resources via mutexes, sharing transports, parent/offspring relationships, etc. The mechanisms of interaction are irrelevant to this specification.

The scheduler architecture assumes that the problem of satisfying scheduling needs can be addressed by managing the allocation of resources to distributable threads. The distributable thread provides a vehicle for carrying scheduling information across the distributed system.

Distributable threads interact with the scheduler at specific scheduling- points, including application calls, locks and releases of resources, and at pre-defined locations within CORBA invocations. The latter are required because CORBA invocations are points at which the distributable thread may transition to another processor, and the scheduling information must be reinterpreted on the new processor.

26.10.1 Scheduler Characteristics

This specification does not assume a single scheduling discipline for Real-time CORBA. Schedulers are developed to implement a particular scheduling discipline or disciplines. Both available products and technical literature abound with examples of schedulers implementing various scheduling disciplines. This specification defines only the interface between the ORB/application and the scheduler, and is intended to foster the development of schedulers that are not dependent on any particular ORB (although a particular scheduler implementation may choose to take advantage of the features of a particular ORB). Note that schedulers will likely be dependent on the underlying operating system, and this specification does not address these operating system interfaces, since they are outside of the scope of CORBA.

This specification addresses schedulers that will optimize execution for the application scheduling needs on a processor-by-processor basis (see Section Distributed System Scheduling, case 2). That is, as the execution of an application distributable thread moves from processor to processor, its scheduling needs are carried along and honored by the scheduler on each processor. This does not preclude the development of schedulers that perform global optimization, but this specification does not specifically address that type of scheduler.

The schedulers considered in relation to this specification will have in common processing stages where they acquire information about the demand for resources, an optional processing stage where they plan the processing schedule (when scheduling, as opposed to dispatching alone, is used), and a processing phase where they affect how threads are dispatched by the operating system. This specification does not impose any requirement on how the scheduler developer defines these processing stages. The specification does define the minimum set of scheduling points (points in time or code when the scheduler will execute).

This specification also provides the scheduler APIs for a small set of scheduling disciplines, including fixed priority, as defined in Real-time CORBA 1.0. This supports application portability for these disciplines.

The current specification does not address full interoperability across scheduler vendor implementations; to achieve this, one would have to define the scheduling discipline, the scheduling parameter elements, and the service context that is used to propagate the scheduling characteristics of the application. The submitters believe that more implementation experience is needed before full interoperability is possible. Therefore, this specification only provides a complete API definition for a limited set of well-understood scheduling disciplines and does not define a standard service context for any scheduling disciplines. Future specifications will define standardized service contexts and the APIs for additional disciplines.

26.10.2 Scheduling Parameter Elements

This specification defines a *scheduling parameter* as a container of potentially multiple values called *scheduling parameter elements*. The scheduling parameter elements are the values needed by a scheduling discipline in order to make scheduling decisions for an application. A scheduling discipline may have no scheduling parameter elements, only one, or several; the number and meaning of the scheduling parameter elements is scheduling discipline specific. A single scheduling parameter, which may contain several scheduling parameter elements, is associated with an executing thread via the **begin_scheduling_segment** operation. A thread executing outside the context of a scheduling segment has no scheduling parameter associated with it and is scheduled by the native scheduling of the operating system, typically priority based.

Some scheduling disciplines will acquire the information about application resource and scheduling requirements at system/application design time (static scheduling); these schedulers typically would load the resulting scheduling information into a data structure that is accessed at run time. Other schedulers are intended to react to dynamic runtime system demands (dynamic scheduling). These cases represent different scheduler “interaction styles.” The interaction style will depend on the

scheduler implementation and, possibly, on the particular scheduling discipline. This specification addresses provides a general scheduler interface that can be used by either style of scheduler interactions.

This specification also allows various types of interactions for static scheduling. The specific approach to be used will be discipline-specific. For example, the application may provide its scheduling parameter elements, and the associated names, in advance so that the scheduler can store them internally; this could be done during some form of application initialization. Alternatively, the application can provide scheduling parameter elements each time it invokes scheduler operations.

The specific information needed by a scheduler will depend on which discipline(s) it implements. For example, simple deadline scheduling may need only the thread's deadline and the amount of CPU time that the thread will consume. Another discipline might utilize relative importance as one of its inputs. This specification has defined a standard interface for passing a set of scheduling discipline-specific information to a scheduler via the elements of a scheduling parameter. The definition of the structure, types, and the handling of these scheduling parameter elements is scheduling discipline-specific. The elements are only defined for the subset of scheduling disciplines provided in this specification.

26.10.3 Pluggable Scheduler and Interoperability

This specification provides a "pluggable" scheduler. A particular ORB in the system may have any scheduler installed, or may have no scheduler. If an ORB has a scheduler installed, all applications run on that ORB are "under the purview" of that scheduler.

Application components may interoperate, in the context of a particular scheduling discipline, as long as their ORBs have compatible schedulers installed (meaning that the schedulers implement the same discipline, and follow a CORBA standard for that discipline) and the scheduler implementations use a compatible service context. As noted above, the current specification does not define any standard service contexts for scheduler interoperability, although future revisions are anticipated in this area.

A scheduler may choose to support multiple disciplines, but this specification does not address how different scheduling disciplines might interact. This may also be addressed in future revisions.

26.10.4 Distributable Threads

A distributable thread (see Section 26.9, "Distributable Thread," on page 26-11) is the fundamental abstraction of application execution in this specification. A distributable thread incorporates the sequence of actions associated with a user-defined portion of the application that may span multiple processing nodes, but that represents a single logical thread of control. Distributed applications will typically be constructed as several distributable threads that execute logically concurrently.

More precisely, a distributable thread is the locus of execution between points in the application that are significant to the application developer, and it carries the scheduling context of the application from node to node as control passes through the system via CORBA requests and replies. It might encompass part of the execution of a local (or native) thread or multiple threads executing in sequence on one or more processors. If it encompasses multiple threads, then it also encompasses the various phases; that is, "in-transit", "static", "active", etc., which might occur as the locus of execution moves among threads.

A distributable thread may have a scheduling parameter containing multiple element values associated with it. These scheduling parameter elements become the scheduling control factors for the distributable thread and are carried with the distributable thread via CORBA requests and replies. Scheduling parameter elements can be associated with a thread by the application invoking the **begin_scheduling_segment** or **update_scheduling_segment** operations (see Section 26.10.6, “Scheduling Segments, Parameter Elements, and Schedulable Entities,” on page 26-19). The application may call the `spawn` operation to create a distributable thread and a corresponding native thread in the current processor and associate scheduling parameter elements with it.

A distributable thread has at most one head (execution point) at any moment in time. If there is a branch of control, as occurs with a CORBA oneway invocation, the originating distributable thread remains at the client and continues execution (as long as it remains the most eligible). A new distributable thread is implicitly created to process each oneway invocations.

Each distributable thread has a globally unique id within the system, which can be accessed via the **get_current_id** operation. The distributable thread id can be used to obtain a reference to a distributable thread, via the **lookup** operation. This reference can then be used to cancel that distributable thread, via the **cancel** operation. The **cancel** operation results in a CORBA::THREAD_CANCEL system exception being raised in the cancelled distributable thread.

26.10.5 Implicit Forking and Joining

Typically, an intrinsic part of any concurrency model is the semantics for the creation of new execution contexts, or *forking*, and the synchronization of multiple execution contexts, or *joining*.

Explicit forking is provided for in this specification by the `spawn` operation. Due to time constraints explicit joining was not provided by this specification. Future finalizations and revision task forces are encouraged to provide for this capability.

Certain aspects of the core CORBA programming model and the programming model of various CORBA services introduce the implicit forking of distributable threads. One example in the core CORBA specification is *oneway* invocations if made with a synchronization scope of **SYNC_NONE** or **SYNC_WITH_TRANSPORT**. This occurs because the distributable thread making the invocation is unblocked before the

operation on the servant executes. Applications may optionally associate an "implicit scheduling parameter" for a distributable thread that is associated with any implicitly created distributable threads created from that distributable thread.

When a distributable thread executing a scheduling segment implicitly forks another distributable thread, the forked distributable thread's scheduling parameter is determined as follows:

- If the implicit scheduling parameter in set for the innermost scheduling segment of the forking distributable thread then the ORB must use this value in implicitly forking any distributable threads.
- Otherwise, the ORB must use the operative scheduling parameter of the innermost scheduling segment for the implicit forking of any distributable threads.

As with forking, there are certain aspects of the core CORBA programming model and the programming model of various CORBA services that introduce the implicit joining of distributable threads. An example of an implicit join is the polling mode introduced by asynchronous messaging. This occurs because the distributable thread calling the poll operation can wait to "join up with" the distributable thread that ran the operation on the servant to get the results of the asynchronous invocation. Note that the initial asynchronous invocation call is an implicit fork that results in the distributable thread used to run the operation on the servant.

When a distributable thread executing a scheduling segment implicitly joins another distributable thread, there is neither inheritance nor propagation of either distributable thread's scheduling parameter to the other distributable thread.

26.10.6 Scheduling Segments, Parameter Elements, and Schedulable Entities

In this specification, distributable threads consist of one or more (potentially nested) *scheduling segments*. Within a distributable thread, scheduling segments can be sequential and/or nested. Nesting creates *scheduling scopes*.

Each scheduling segment represents a sequence of control flow with which a particular set of scheduling parameter elements is associated. A scheduling segment is delineated by **begin_scheduling_segment** and **end_scheduling_segment** statements in the code. The application may use the segment name on the end statement, as an error check. The scheduling parameter associated with a distributable thread may be updated with a call to **update_scheduling_segment**.

At runtime, a scheduling segment has a single starting point, and a single ending point (although it could be coded with multiple possible ending points, during execution only one ending point can be invoked). Segments may span processor boundaries. This specification places no restrictions on the placement of **begin_scheduling_segment**'s and **end_scheduling_segment**'s; an **end_scheduling_segment** may occur on a different processor than the **begin_scheduling_segment**, and may even occur somewhere up the chain of CORBA requests.

As a distributable thread moves from object instance to object instance through CORBA invocations, it may extend (and possibly retract) itself through one or more processes or processors. When this happens, the distributable thread may be contending with a new set of distributable threads for resources.

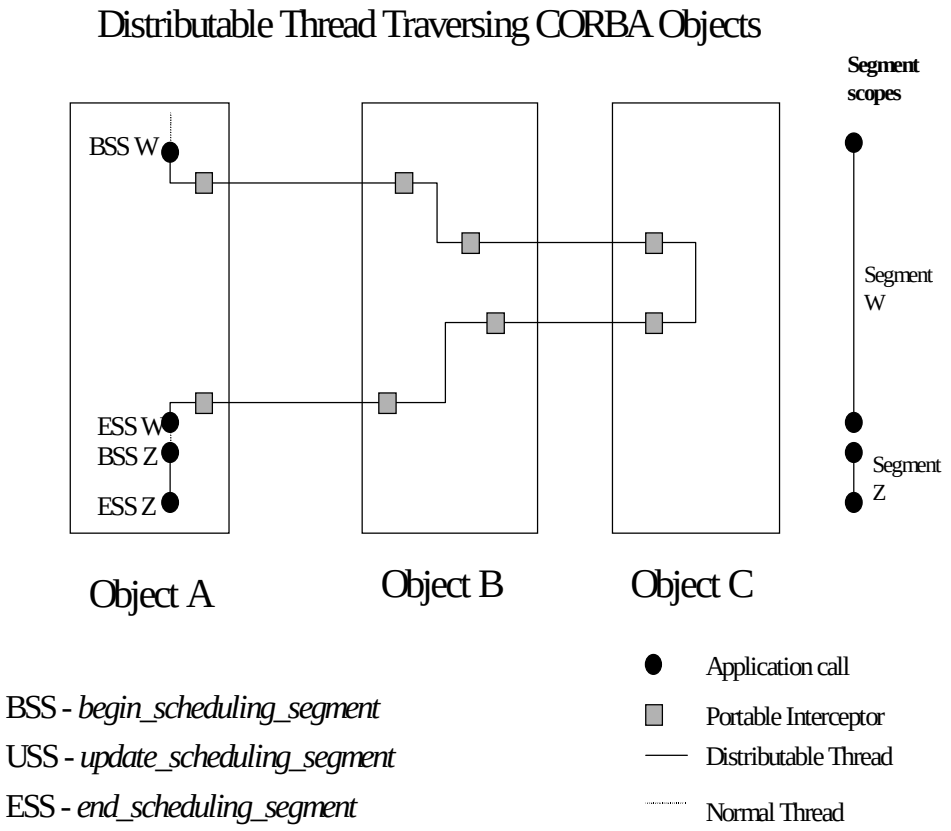

Figure 26-3 A Distributable Thread with Two Sequential Segments

Figure 26-3 illustrates a simple distributable thread which contains two sequential segments. The distributable thread begins in object instance A, with segment W, and traverses object instances B and C before returning to A, where the first segment ends and a new segment (Z) begins. Portable interceptors are invoked each time the distributable thread transitions to another object instance via a CORBA request (on both the client and servant side) and again as the distributable thread returns. Note that these object instances could be on different processors.

Suppose the scheduling discipline is Earliest Deadline First, which implies that the illustrated distributed thread must (implicitly) carry its deadline along as it progresses through the various processor environments. Further, assume that the scheduling discipline calls for scheduling segments that have missed their deadline to be terminated. This last condition implies that the scheduler must be maintaining a list of

deadlines. The **begin_scheduling_segment**, **update_scheduling_segment**, and **end_scheduling_segment** operations serve to enter, update or remove deadlines, but the scheduler must also address what happens when a set deadline expires.

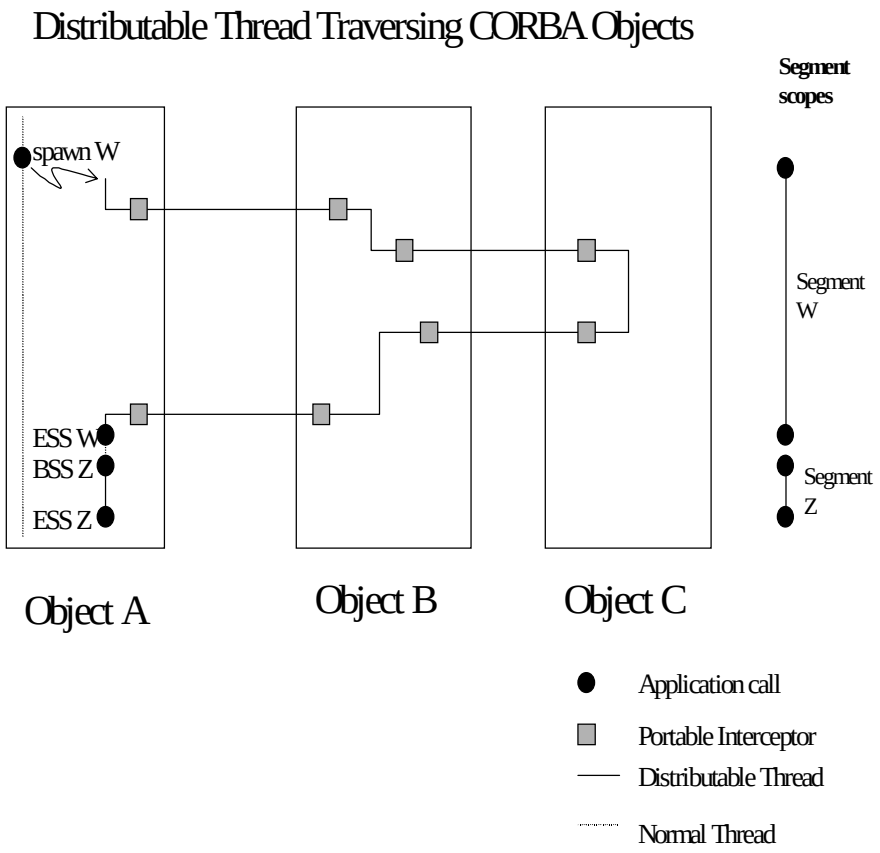

Figure 26-4 A Distributable Thread Created by a Spawn Operation

Figure 26-4 illustrates the use of a *spawn* to create a distributable thread. Note that the spawn also serves as the beginning for the initial segment (W) of the distributable thread.

Some scheduling disciplines may support the nesting of scheduling segments, which permits independently developed software components to define their own scheduling segments. The component would create an additional scheduling segment by embedding one or more pair of calls to **begin_scheduling_segment** and **end_scheduling_segment**. The handling of unspecified parameter elements (defaulting) is discipline-specific. In some cases, unspecified elements will use the values from the next outer segment (if any). In other cases, predefined or application defined default values might be used.

Each **begin_scheduling_segment** provides a new set of scheduling parameter elements for the distributable thread. If the distributable thread is already in a segment, these new parameter elements will replace the current set until a matching **end_scheduling_segment** occurs. An **end_scheduling_segment** statement causes the distributable thread to return to the previous scheduling parameter (if any). Thus, a distributable thread may contain multiple scheduling segments that are executed sequentially, each of which may contain nested segments. This specification does not place any limits on the level of nesting that a scheduling discipline will support.

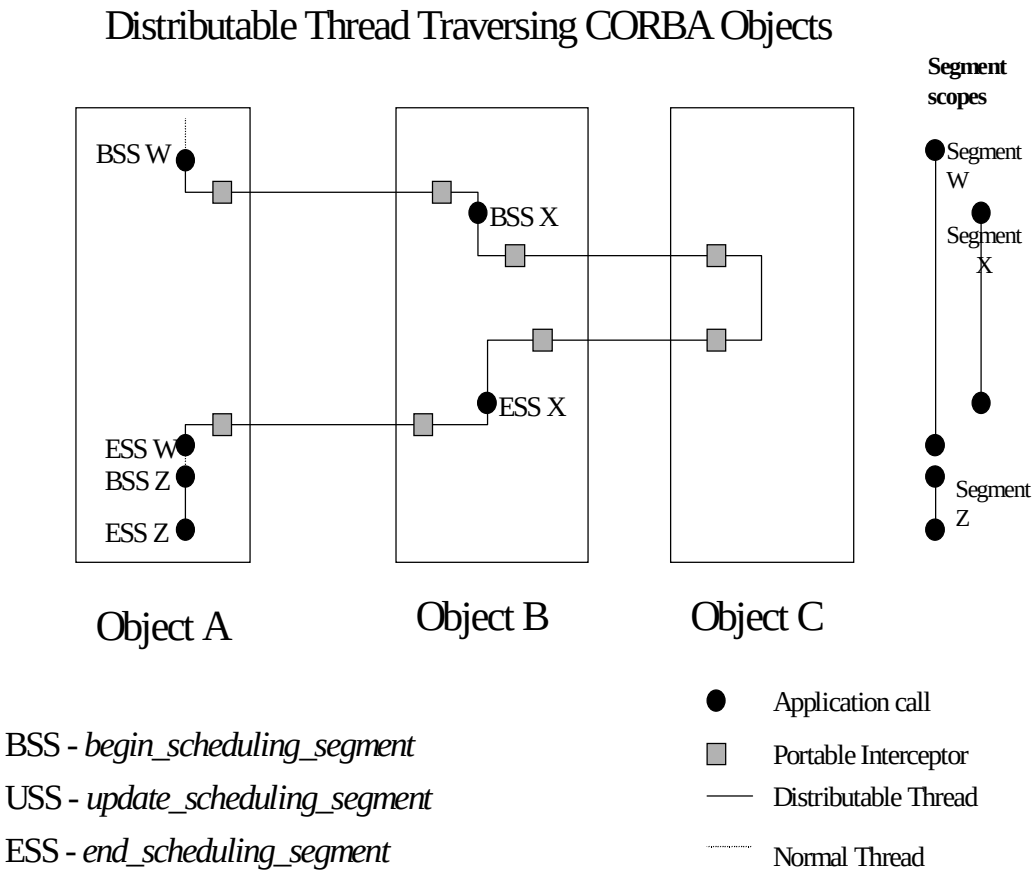

Figure 26-5 Distributable Thread with Nested Segments

Figure 26-5 illustrates segment nesting. In this case, segment X is nested within segment W. At the point where segment X begins, the scheduling context of segment W is logically pushed onto a stack, and segment W's scheduling parameter elements are used for the distributable thread. When segment X ends, the distributable thread returns to the scheduling parameter elements for segment W.

In the case of EDF, all of these segments involve the requirement that they complete by some deadline, but they would probably be different deadlines. In the case of nested segments (W, X, and Y) the tightest deadline may come from any of the segments.

A distributed thread executing in a single object instance may, at different times, have different deadlines. Note that where the distributed thread first executes in object instance B its deadline will be the deadline for segment W. However, as soon as segment X begins, the deadline must be selected from the tighter of the outer (W) or inner (X) scheduling segment.

It is expected that each instance of the scheduler must monitor the time constraints of every distributed thread that is currently traversing its node.

A scheduling parameter element that is created in one object instance must be considered in other object instances as the distributed thread passes through them. In the illustration, the deadline established in object instance A must be considered with respect to all other deadlines that exist in the domain of object B, and similarly as the distributed thread extends to object C.

How a scheduler addresses distributed dynamic scheduling is implementation dependent, but it is likely that the features of the portable interceptor will be required. By requiring use of an interceptor that targets the scheduler for the outgoing and incoming sides of the connection at both the client and server sides, the scheduler can address these characteristics. A client-side outgoing interceptor can address moving the deadline compliance monitoring while the associated server side incoming interceptor can address the continuing deadline compliance monitoring and distributed thread scheduling with respect to the server side workload.

The application may also invoke the scheduler within a segment, either to allow the scheduler to notify the application if it has had a scheduling failure (such as a missed deadline), or to modify the current segment's scheduling parameter elements. This is done via the **update_scheduling_segment** operation. The update operation allows the application to occasionally check in with the scheduler, and can also be used to change scheduling parameter elements dynamically, without creating a new segment.

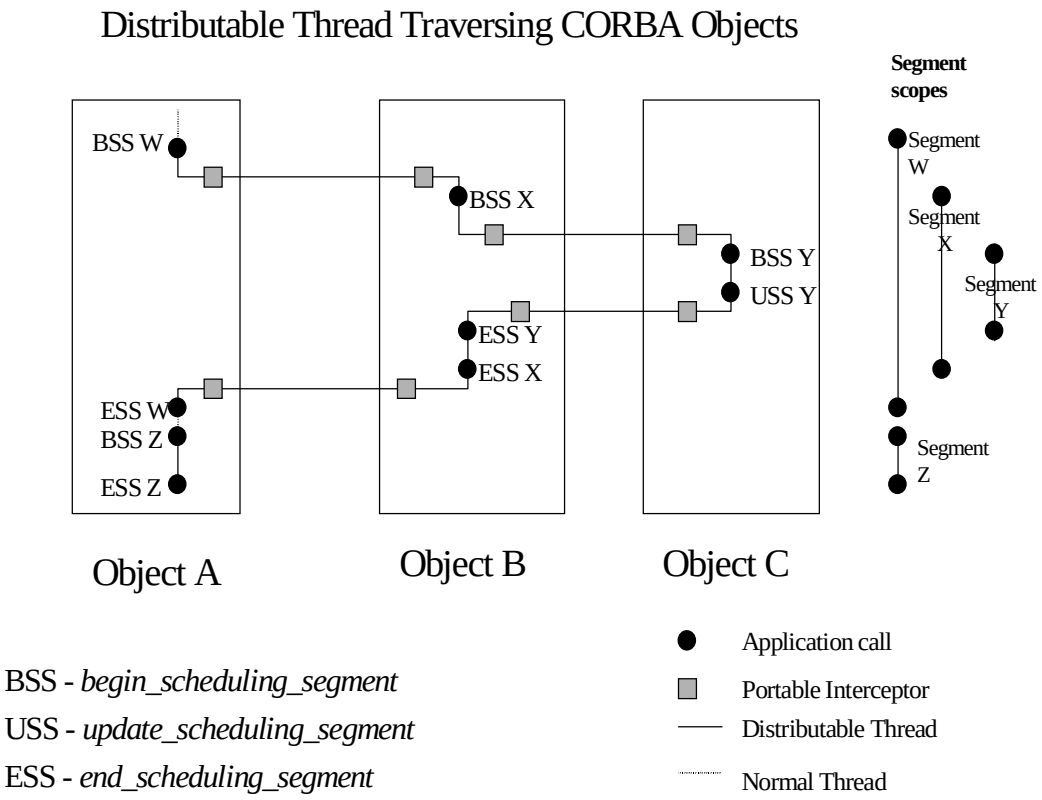

Figure 26-6 Distributable Thread with Nested Segments

Figure 26-6 illustrates the remaining features of scheduling segments, namely the use of multi-level nesting, updates, and the flexible placement of ends. Note that there are two levels of nesting within segment W. In this example, an **update_scheduling_segment** is called within segment Y. Any exceptions for the distributable thread could be delivered at this point, rather than waiting for the next portable interceptor call or the **end_scheduling_segment**. In addition, the application could provide new scheduling parameter elements on the update, without returning to the next upper scheduling scope. Note also, this in this example, segment Y is begun in object instance C, but ended in object instance B, which was the invoker of object instance C.

The application can obtain a list of the current scheduling segment names, innermost scope first, via the **current_scheduling_segment_names** operation.

26.10.7 Scheduling Points

There are a number of *scheduling points*, which are points in time and/or code at which the scheduler is run and may result in an alteration of the current schedule. These include all begins and ends, access to shared resources, and points at which control

transfers between processing nodes (i.e., CORBA requests). Because these scheduling points may result in schedule changes, they may also be a point at which dispatching occurs.

The following set of scheduling points is defined:

- Creation of a distributable thread (via **begin_scheduling_segment** or **spawn**)
- Termination or completion of a distributable thread
- **begin_scheduling_segment**
- **update_scheduling_segment**
- **end_scheduling_segment**
- A CORBA operation invocation, specifically the request and reply interception points provided in the Portable Interceptor specification
- Creation of a resource manager
- Blocking on a request for a resource via a call to **RTScheduling::ResourceManager::lock** or **RTScheduling::ResourceManager::try_lock**
- Unblocking as a result of the release of a resource via a call to **RTScheduling::ResourceManager::unlock**

26.10.8 Schedule-Aware Resources

This specification permits the application to create a scheduler-aware resource locally via the **create_resource_manager** operation in a **ResourceManager**; these resources can have scheduling information associated with them via the **set_scheduling_parameter** operation. For example, a servant thread could have a priority ceiling if the application were using fixed priority scheduling. The scheduler will run when these resources are locked or released, so that the scheduling discipline is maintained.

Any scheduling information associated with these resources is scheduling discipline-specific.

26.10.9 Exceptions

This specification defines the following exceptions related to scheduling:

- CORBA::SCHEDULER_FAULT – this indicates that the scheduler itself has experienced an error.
- CORBA::SCHEDULE_FAILURE – this indicates that the distributable thread has violated the constraints of its scheduling parameter. For example, this exception could occur when a deadline has been missed or a segment has used more than its allowed CPU time.

- CORBA::THREAD_CANCELLED – indicates that the distributable thread receiving the exception has been cancelled. This may occur because a distributable thread cancels another distributable thread thereby causing the CORBA::THREAD_CANCELLED exception to get raised at the subsequent head of the cancelled distributable thread.
- RTScheduling:: UNSUPPORTED_SCHEDULING_DISCIPLINE – indicates that the scheduler was passed a scheduling parameter inappropriate for the scheduling discipline(s) supported by the current scheduler.

26.10.10 Summary

An application consists of one or more distributable threads (as well as possibly local processor threads which are not part of distributable threads). Each distributable thread will execute through one or a series of (distributed) scheduling segments, including some that may have nested segments. These segments represent regions of execution that have their own scheduling parameter elements. Within these scheduling segments, additional calls may be made to alter the scheduling parameter elements and/or to just allow the scheduler to run.

Distributable threads may evolve from application threads, due to a **begin_scheduling_segment** operation, a one-way operation, or be generated by `spawn` operations. Distributable threads may be cancelled by another distributable thread, and cancelled distributable threads will be notified of the cancellation via an exception.

These distributable threads may share local resources utilizing resource manager **lock**, **try_lock**, and **unlock** operations. These operations are schedule-respecting.

Section IV - Scheduler Interoperability and Portability

26.11 Scheduler Interoperability

A CORBA ORB supporting dynamic scheduling will interoperate with an ORB that does not support this capability. The scheduling parameter for a distributable thread is passed to the other ORB in the service context field and the other ORB can ignore them.

An ORB conformant with Real-time CORBA 1.0 will interoperate with an ORB compliant with this specification in the functional sense (i.e., without regard to timeliness). An ORB compliant with this specification that has no scheduler installed is fully interoperable in both terms of functionality and timeliness. If a scheduler is installed then timeliness characteristics of the resulting system will depend on the installed scheduler and its backwards compatibility with the Real-time CORBA 1.0 fixed priority scheduling.

26.12 Scheduler Portability

This specification addresses the issue of portability between the ORB and scheduler and also between the application and the scheduler. This specification provides that capability in that it makes the ORB/scheduler interfaces available to applications.

26.13 Dynamic Scheduling Interoperation

This specification does not address interoperation between different dynamic scheduler implementations or between different scheduling disciplines.

Dynamic Scheduling is an extension of and modification to the RT CORBA specification. Application functions that are scheduled using the fixed priority methods will interoperate with dynamic scheduling tasks. This specification offers the application developer several options with regard to mixed mode operations. For example, a band of priorities can be reserved for dynamically scheduled activities. That band may be located at the high or low end of the priority range or it may be placed in the middle of the priority band. When activities have a priority higher than the dynamic scheduling band then dynamic scheduled activities will only run during what would otherwise be idle time. When dynamic scheduling is given top priority the scheduler resources might be dedicated to some activities while the remainder of the activities are dispatched during periods when the dynamically scheduled activities are not ready to execute.

Schedulers may be constructed so that dynamic scheduling systems can provide services to non-dynamically scheduled CORBA client applications. Requests from such an client would be treated as any processing that occurs without a scheduling parameter set. When dynamically scheduled clients make requests to non-dynamically scheduled servants then the added information carried in the service contexts is ignored. The request is valid but is not dynamically scheduled.

Section V - Dynamic Scheduling Interfaces

26.14 ThreadAction Interface

26.14.1 do Operation

26.14.1.1 IDL

```
module RTScheduling
{

...
    local interface ThreadAction
    {
        void do(in CORBA::VoidData data);
```

```
    };
...
};
```

26.14.1.2 Semantics

The **ThreadAction** interface is used to provide an entry point for newly spawned distributable threads. The **ThreadAction** interface serves as a parent type for user implemented **ThreadAction** objects. The **ThreadAction::do** operation by default does nothing. User written overrides of the **do** operation are expected execute the application's thread-specific actions.

26.15 RTScheduling::Current Interface

The **RTScheduling::Current** interface is derived from **RTCORBA::Current**. An ORB that implements this specification returns a reference from a call to **CORBA::ORB::resolve_initial_references** with the **"RTCurrent"** value passed via the *identifier* parameter that can be narrowed to an **RTScheduling::Current** reference.

26.15.1 spawn Operation

26.15.1.1 IDL

```
module RTScheduling
{
    ...
    local interface Current
        : RTCORBA::Current
    {
        ...
        DistributableThread
            spawn
                (in ThreadAction     start,
                 in unsigned long    stack_size,
                     // zero means use the O/S default
                 in RTCORBA::Priority base_priority);
        ...
    };
    ...
};
```

26.15.1.2 Semantics

The **spawn** operation creates a new O/S thread and makes that thread a distributable thread with a stack size at least as large as the value passed in the **stack_size** parameter. The initial CORBA base priority is the value passed by the **base_priority** parameter. The new distributable thread calls the do operation on the **ThreadAction** object passed via the start parameter.

26.15.2 UNSUPPORTED_SCHEDULING_DISCIPLINE Exception

26.15.2.1 IDL

```
module RTScheduling
{
    ...
    local interface Current : RTCORBA::Current
    {
        ...
        exception UNSUPPORTED_SCHEDULING_DISCIPLINE {};
        ...
    };
    ...
};
```

26.15.2.2 Semantics

The UNSUPPORTED_SCHEDULING_DISCIPLINE exception is raised when a scheduling parameter argument isn't appropriate for the installed scheduler instance.

26.15.3 begin_scheduling_segment Operation

26.15.3.1 IDL

```
module RTScheduling
{
    ...
    local interface Current : RTCORBA::Current
    {
        ...
        void begin_scheduling_segment
            (in string      name,
             in CORBA::Policy sched_param,
             in CORBA::Policy implicit_sched_param)
            raises UNSUPPORTED_SCHEDULING_DISCIPLINE);
        ...
    };
    ...
```

```
};
```

26.15.3.2 Semantics

The **begin_scheduling_segment** operation raises the RTScheduling::UNSUPPORTED_SCHEDULING_DISCIPLINE exception when the **scheduling_parameter** argument didn't have an appropriate value for the active scheduling discipline.

The **begin_scheduling_segment** operation raises the CORBA::SCHEDULE_FAILURE exception when the scheduling segment failed its schedule.

The **begin_scheduling_segment** operation raises the CORBA::SCHEDULER_FAULT exception when the scheduler has had internal error.

The **begin_scheduling_segment** operation raises the CORBA::THREAD_CANCELLED exception when the distributable thread was cancelled.

The **begin_scheduling_segment** operation raises the CORBA::BAD_PARAM exception when the **scheduling_parameter**, any elements of the scheduling parameter, or the name parameter was invalid for the installed scheduler.

The **begin_scheduling_segment** operation begins a scheduling segment, and converts the currently executing thread into a distributable thread, if it is not already one. A scheduling segment is a window of execution where a distributable thread is executing a particular region of code. The scheduler conditions execution of a particular scheduling segment using the passed **scheduling_parameter** argument, until a **begin_scheduling_segment**. **update_scheduling_segment**, or **end_scheduling_segment** is encountered.

The name parameter provides identification for the region of code that comprises the scheduling segment. Some schedulers may support nesting of scheduling segments. If a scheduler does not support nesting of scheduling segments this operation raises CORBA::SCHEDULE_FAILURE.

A **scheduling_parameter** contains elements that are a value or set of values appropriate for the active scheduling discipline. The **scheduling_parameter** used by the scheduler and set by the application.

The requirements for the "**scheduling_parameter**" and "name" parameters are dependant on both the scheduling discipline defined, and on the interaction style supported by the scheduler. It is expected that at least one these parameters ("**scheduling_parameter**" or "name") is a non-null argument.

In addition, the **begin_scheduling_segment** operation provides a scheduling point for the scheduler and gives the scheduler an opportunity to cancel a distributable thread by raising the CORBA::THREAD_CANCELLED exception while is executing in a scheduling segment.

26.15.4 update_scheduling_segment Operation

26.15.4.1 IDL

```
module RTScheduling
{
    ...
    local interface Current : RTCORBA::Current
    {
        ...
    void update_scheduling_segment
        (in string      name,
         in CORBA::Policy sched_param,
         in CORBA::Policy implicit_sched_param)
        raises (UNSUPPORTED_SCHEDULING_DISCIPLINE);
        ...
    };
    ...
};
```

26.15.4.2 Semantics

The **update_scheduling_segment** operation raises the RTScheduling::DistributableThread::UNSUPPORTED_SCHEDULING_DISCIPLINE exception when the **scheduling_parameter** argument didn't have an appropriate value for the active scheduling discipline.

The **update_scheduling_segment** operation raises the CORBA::SCHEDULE_FAILURE exception when the scheduling segment failed its schedule.

The **update_scheduling_segment** operation raises the CORBA::SCHEDULER_FAULT exception when the scheduler has had internal error.

The **update_scheduling_segment** operation raises the CORBA::THREAD_CANCELLED exception when the distributable thread was cancelled.

The **update_scheduling_segment** operation raises the CORBA::BAD_PARAM exception when the **scheduling_parameter** or any elements of the scheduling parameter are invalid for the installed scheduler.

The **update_scheduling_segment** operation provides the scheduler with a scheduling point and provides an opportunity for the scheduler to check for a scheduling failure. In addition, the **update_scheduling_segment** operation gives the scheduler an opportunity to raise the CORBA::THREAD_CANCELLED exception within a distributable thread while it is executing in a scheduling segment.

The **update_scheduling_segment** operation should only be called inside of a scheduling segment. A call to the **update_scheduling_segment** operation outside of a scheduling segment raises CORBA::SCHEDULE_FAILURE.

Any non-null value passed via the **scheduling_parameter** parameter allows an application to request that a scheduler update the scheduling parameter or the implicit scheduling parameter, or both, associated with enclosing scheduling segment. A null value indicates to the scheduler that there it should not update scheduling parameter associated with the enclosing scheduling segment.

26.15.5 end_scheduling_segment Operation

26.15.5.1 IDL

```
module RTScheduling
{
    ...
    local interface Current : RTCORBA::Current
    {
        ...
        void end_scheduling_segment(in string name);
        ...
    };
    ...
};
```

26.15.5.2 Semantics

The **end_scheduling_segment** operation raises the CORBA::SCHEDULE_FAILURE exception when the scheduling segment failed its schedule.

The **end_scheduling_segment** operation raises the CORBA::SCHEDULER_FAULT exception when the scheduler has had internal error.

The **end_scheduling_segment** operation raises the CORBA::THREAD_CANCELLED exception when the distributable thread was cancelled.

The **end_scheduling_segment** operation raises the CORBA::BAD_PARAM exception when the name parameter was invalid for the installed scheduler.

The **end_scheduling_segment** operation ends a scheduling segment. Each call to a **end_scheduling_segment** operation should match a call to begin_scheduling_segment made in the same distributable thread. If **end_scheduling_segment** is called in a distributable thread that does not have a matching call to **begin_scheduling_segment** raises CORBA::SCHEDULE_FAILURE.

The **end_scheduling_segment** operation provides the scheduler with a scheduling point and provides an opportunity for the scheduler to check for a scheduling failure.

If a non-null string is passed via the name parameter then the scheduler can verify the name with the name passed in the corresponding **begin_scheduling_segment** call. If a null string is passed then no verification takes place.

After an **end_scheduling_segment** operation, the distributable thread is either operating with the scheduling parameter of the next outermost scheduling segment scope. If this operation is performed at the outermost scope, the result is that the processing for that thread reverts back to the fixed priority scheduling where the active thread priority is the sole determinant of the threads eligibility for execution.

26.15.6 Id Related Operations

26.15.6.1 IDL

```
module RTScheduling
{
    ...
    local interface Current : RTCORBA::Current
    {
        ...

        IdType get_current_id();
        // returns id of thread that is running

        DistributableThread lookup(in IdType id);
            // returns a null reference if
            // the distributable thread is
            // not known to the local scheduler

        typedef sequence<octet> IdType;

        readonly attribute IdType id;
            // a globally unique id
        ...
    };
    ...
};
```

26.15.6.2 Semantics

Each distributable thread has a globally unique id within the system, which can be accessed via the **get_current_id** operation. The distributable thread id can be used to obtain a reference to a distributable thread, via the `lookup` operation. This reference can then be used to cancel that distributable thread, via the

RTScheduling::DistributableThread::cancel operation. This **cancel** operation results in a CORBA::THREAD_CANCEL system exception being raised at the head of the cancelled distributable thread.

26.15.7 scheduling_parameter and implicit_scheduling_parameter Attributes

26.15.7.1 IDL

```
module RTScheduling
{
    ...
    local interface Current : RTCORBA::Current
    {
        ...
        readonly attribute CORBA::Policy
                        scheduling_parameter;
        readonly attribute CORBA::Policy
                        implicit_scheduling_parameter;
        ...
    };
    ...
};
```

26.15.7.2 Semantics

Each distributable thread has a current scheduling policy if it is operating in a scheduling segment (and a null scheduling policy otherwise). The **scheduling_parameter** attribute returns the scheduling parameter for the innermost segment name.

The **implicit_scheduling_parameter** attribute returns the implicit scheduling parameter as last set by a **begin_scheduling_segment** or **update_scheduling_segment** call for the current distributable thread.

If the distributable thread is executing outside the context of the scheduling segment then a null reference is returned from either of this attributes.

26.15.8 current_scheduling_segment_names Attribute

26.15.8.1 IDL

```
module RTScheduling
{
    ...
    local interface Current : RTCORBA::Current
    {
        ...
```

```
        typedef sequence<string> NameList;

        readonly attribute NameList
                    current_scheduling_segment_names;
            // Ordered from innermost segment name
            // to outmost segment name
        ...
    };
    ...
};
```

The **current_scheduling_segment_names** attribute returns a list of the current scheduling segment names, innermost scope first.

26.16 RTScheduling::ResourceManager Interface

26.16.1 IDL

```
module RTScheduling
{
    ...
    local interface ResourceManager : RTCORBA::Mutex
    {
    };
    ...
};
```

26.17 RTScheduling::DistributableThread Interface

26.17.1 IDL

```
module RTScheduling
{
    ...
    local interface DistributableThread
    {
        void cancel();
            // raises CORBA::OBJECT_NOT_FOUND if
            // the distributable thread is
            // not known to the scheduler
    };
    ...
};
```

26.17.2 cancel Operation

The **cancel** operation causes the CORBA::THREAD_CANCELLED exception to be raised at the head of the distributable thread. Note that while the **DistributableThread** is a local interface the head of the distributable thread may not be executing within the same address space as thread calling **cancel**.

26.18 RTScheduling::Scheduler Interface

The scheduler interface is a local interface with the semantics of an abstract interface. Its purpose is to delineate the core interface of a scheduler such that the **Scheduler** interface is used as a parent interface of a scheduler plug-in.

An object reference to the currently installed scheduler is obtained by calling **CORBA::ORB::resolve_initial_references** with the *identifier* parameter set to the value "**RTScheduler**." If no scheduler is installed a null object reference is returned.

26.18.1 Scheduler:: INCOMPATIBLE_SCHEDULING_DISCIPLINES Exception

26.18.1.1 IDL

```
module RTScheduling
{
    ...
    local interface Scheduler
    {
        ...
        exception INCOMPATIBLE_SCHEDULING_DISCIPLINES {};
        ...
    };
    ...
};
```

26.18.2 Scheduler::scheduling_policies Attribute

26.18.2.1 IDL

```
module RTScheduling
{
    ...
    local interface Scheduler
    {
        ...
        attribute CORBA::PolicyList
            scheduling_policies;
        ...
```

```
    };
    ...
};
```

26.18.3 Scheduler::poa_polices Attribute

26.18.3.1 IDL

```
module RTScheduling
{
    ...
    local interface Scheduler
    {
        ...
        readonly attribute CORBA::PolicyList poa_policies;
        ...
    };
    ...
};
```

26.18.3.2 Semantics

The **scheduling_policies** attribute allows the ORB to request the list of POA policies that the scheduler requires to be applied to all POA's associated with this ORB. A null list is an acceptable result value.

26.18.4 Scheduler::scheduling_discipline_name Attribute

26.18.4.1 IDL

```
module RTScheduling
{
    ...
    local interface Scheduler
    {
        ...
        readonly attribute string
            scheduling_discipline_name;
        ...
    };
    ...
};
```

26.18.4.2 Semantics

A simple string containing the textual name of the scheduling discipline for use by both the ORB and application.

26.18.5 Scheduler::create_resource_manager Operation

26.18.5.1 IDL

```
module RTScheduling
{
    ...
    local interface Scheduler
    {
        ...
        ResourceManager
            create
                (in string       name,
                 in CORBA::Policy scheduling_parameter);
        ...
    };
    ...
};
            // raises (CORBA::BAD_OPERATION, CORBA::BAD_PARAM,
CORBA::NO_RESOURCES);
```

26.18.5.2 Semantics

Used by application developers to create a scheduler aware resource protection primitive, and associated a name with the resource.

26.18.6 Scheduler::set_scheduling_parameter Operation

26.18.6.1 IDL

```
module RTScheduling
{
    ...
    local interface Scheduler
    {
        ...
        void set_scheduling_parameter
            (inout PortableServer::Servant resource,
             in    string       name,
             in    CORBA::Policy  scheduling_parameter);
        ...
    };
```

```
    ...
};
```

26.18.6.2 *Semantics*

The **set_scheduling_parameter** operation associates the supplied scheduling parameter and name parameter with the supplied servant resource.

The "**resource**" parameter is a required parameter.

The requirements for the "**scheduling_parameter**" and "**name**" parameters are scheduling discipline defined. It is expected that at least one these parameters ("**scheduling_parameter**" or "**name**") is a non-null argument.

This is useful for schedulers that associate some scheduling information with a shared resource. An example of this type of scheduler would be a fixed priority scheduler that uses some form of priority ceiling protocol.

A.1 Conformance

This specification makes changes to the Real-time CORBA 1.0 specification that an implementation must conform to comply with the Real-time CORBA 1.0 specification. However, the changes to the Real-time CORBA 1.0 specification that are required of existing implementations are minor and only affect ORBs compliant with this specification.

While, the implementation of this specification is optional for implementations of the Real-time CORBA 1.0 specification the opposite is not true. For an ORB to comply with this specification it must conform to the Real-time CORBA 1.0 specification. In particular, an ORB that conforms to this specification must implement the fixed priority scheduling of the Real-time CORBA 1.0 specification when no scheduler is installed.

The implementation of the basic Dynamic Scheduling infrastructure (i.e., the implementation of all interfaces and associated semantics not associated with a particular scheduler) is the most basic form of compliance with this specification.

Implementing a scheduler that conforms to one of the scheduling disciplines in this specification (i.e., it implements of all interfaces and associated semantics for that scheduling discipline) is an optional and separate compliance point for a conforming implementation of the basic Dynamic Scheduling infrastructure. Nesting of scheduling segments is not required feature for the conformance of a scheduler that implements any one of the specified scheduling disciplines in this specification.

B.1 Extensions to core CORBA

B.1.1 Additional System Exceptions

This specification adds three system exceptions to core CORBA:

- CORBA::SCHEDULER_FAULT
- CORBA::SCHEDULE_FAILURE
- CORBA::THREAD_CANCELLED

See Section 26.10.9, “Exceptions,” on page 26-25 for more details.