The Portable Object Adaptor 11

The Portable Object Adaptor chapter has been updated based on CORE changes from ptc/98-09-04.

This chapter describes the Portable Object Adapter, or POA. It presents the design goals, a description of the abstract model of the POA and its interfaces, followed by a detailed description of the interfaces themselves.

Contents

This chapter contains the following sections.

Section Title	Page
“Overview”	11-1
“Abstract Model Description”	11-2
“Interfaces”	11-13
“IDL for PortableServer module”	11-41
“UML Description of PortableServer”	11-48
“Usage Scenarios”	11-49

11.1 Overview

The POA is designed to meet the following goals:

- Allow programmers to construct object implementations that are portable between different ORB products.

- Provide support for objects with persistent identities. More precisely, the POA is designed to allow programmers to build object implementations that can provide consistent service for objects whose lifetimes (from the perspective of a client holding a reference for such an object) span multiple server lifetimes.
- Provide support for transparent activation of objects.
- Allow a single servant to support multiple object identities simultaneously.
- Allow multiple distinct instances of the POA to exist in a server.
- Provide support for transient objects with minimal programming effort and overhead.
- Provide support for implicit activation of servants with POA-allocated Object Ids.
- Allow object implementations to be maximally responsible for an object's behavior. Specifically, an implementation can control an object's behavior by establishing the datum that defines an object's identity, determining the relationship between the object's identity and the object's state, managing the storage and retrieval of the object's state, providing the code that will be executed in response to requests, and determining whether or not the object exists at any point in time.
- Avoid requiring the ORB to maintain persistent state describing individual objects, their identities, where their state is stored, whether certain identity values have been previously used or not, whether an object has ceased to exist or not, and so on.
- Provide an extensible mechanism for associating policy information with objects implemented in the POA.
- Allow programmers to construct object implementations that inherit from static skeleton classes, generated by OMG IDL compilers, or a DSI implementation.

11.2 Abstract Model Description

The POA interfaces described in this chapter imply a particular abstract computational model. This section presents that model and defines terminology and basic concepts that will be used in subsequent sections.

This section provides the rationale for the POA design, describes some of its intended uses, and provides a background for understanding the interface descriptions.

11.2.1 Model Components

The model supported by the POA is a specialization of the general object model described in the OMA guide. Most of the elements of the CORBA object model are present in the model described here, but there are some new components, and some of the names of existing components are defined more precisely than they are in the CORBA object model. The abstract model supported by the POA has the following components:

- *Client*—A client is a computational context that makes requests on an object through one of its references.

- *Server*—A server is a computational context in which the implementation of an object exists. Generally, a server corresponds to a process. Note that *client* and *server* are roles that programs play with respect to a given object. A program that is a client for one object may be the server for another. The same process may be both client and server for a single object.
- *Object*—In this discussion, we use *object* to indicate a CORBA object in the abstract sense, that is, a programming entity with an identity, an interface, and an implementation. From a client's perspective, the object's identity is encapsulated in the object's reference. This specification defines the server's view of object identity, which is explicitly managed by object implementations through the POA interface.
- *Servant*—A servant is a programming language object or entity that implements requests on one or more objects. Servants generally exist within the context of a server process. Requests made on an object's references are mediated by the ORB and transformed into invocations on a particular servant. In the course of an object's lifetime it may be associated with (that is, requests on its references will be targeted at) multiple servants.
- *Object Id*—An Object Id is a value that is used by the POA and by the user-supplied implementation to identify a particular abstract CORBA object. Object Id values may be assigned and managed by the POA, or they may be assigned and managed by the implementation. Object Id values are hidden from clients, encapsulated by references. Object Ids have no standard form; they are managed by the POA as uninterpreted octet sequences.

  Note that the Object Id defined in this specification is a mechanical device used by an object implementation to correlate incoming requests with references it has previously created and exposed to clients. It does not constitute a unique logical identity for an object in any larger sense. The assignment and interpretation of Object Id values is primarily the responsibility of the application developer, although the **SYSTEM_ID** policy enables the POA to generate Object Id values for the application.
- *Object Reference*—An object reference in this model is the same as in the CORBA object model. This model implies, however, that a reference specifically encapsulates an Object Id and a POA identity.

  Note that a concrete reference in a specific ORB implementation will contain more information, such as the location of the server and POA in question. For example, it might contain the full name of the POA (the names of all POAs starting from the root and ending with the specific POA). The reference might not, in fact, actually contain the Object Id, but instead contain more compact values managed by the ORB which can be mapped to the Object Id. This is a description of the abstract information model implied by the POA. Whatever encoding is used to represent the POA name and the Object Id must not restrict the ability to use any legal character in a POA name or any legal octet in an Object Id.

- *POA*—A POA is an identifiable entity within the context of a server. Each POA provides a namespace for Object Ids and a namespace for other (nested or child) POAs. Policies associated with a POA describe characteristics of the objects implemented in that POA. Nested POAs form a hierarchical name space for objects within a server.
- *Policy*—A Policy is an object associated with a POA by an application in order to specify a characteristic shared by the objects implemented in that POA. This specification defines policies controlling the POA's threading model as well as a variety of other options related to the management of objects. Other specifications may define other policies that affect how an ORB processes requests on objects implemented in the POA.
- *POA Manager*—A POA manager is an object that encapsulates the processing state of one or more POAs. Using operations on a POA manager, the developer can cause requests for the associated POAs to be queued or discarded. The developer can also use the POA manager to deactivate the POAs.
- *Servant Manager*—A servant manager is an object that the application developer can associate with a POA. The ORB will invoke operations on servant managers to activate servants on demand, and to deactivate servants. Servant managers are responsible for managing the association of an object (as characterized by its Object Id value) with a particular servant, and for determining whether an object exists or not. There are two kinds of servant managers, called **ServantActivator** and **ServantLocator**; the type used in a particular situation depends on policies in the POA.
- *Adapter Activator*—An adapter activator is an object that the application developer can associate with a POA. The ORB will invoke an operation on an adapter activator when a request is received for a child POA that does not currently exist. The adapter activator can then create the required POA on demand.

11.2.2 Model Architecture

This section describes the architecture of the abstract model implied by the POA, and the interactions between various components. The ORB is an abstraction visible to both the client and server. The POA is an object visible to the server. User-supplied implementations are registered with the POA (this statement is a simplification; more detail is provided below). Clients hold references upon which they can make requests. The ORB, POA, and implementation all cooperate to determine which servant the operation should be invoked on, and to perform the invocation.

Figure 11-1 shows the detail of the relationship between the POA and the implementation. Ultimately, a POA deals with an Object Id and an active servant. By *active servant*, we mean a programming object that exists in memory and has been presented to the POA with one or more associated object identities. There are several ways for this association to be made.

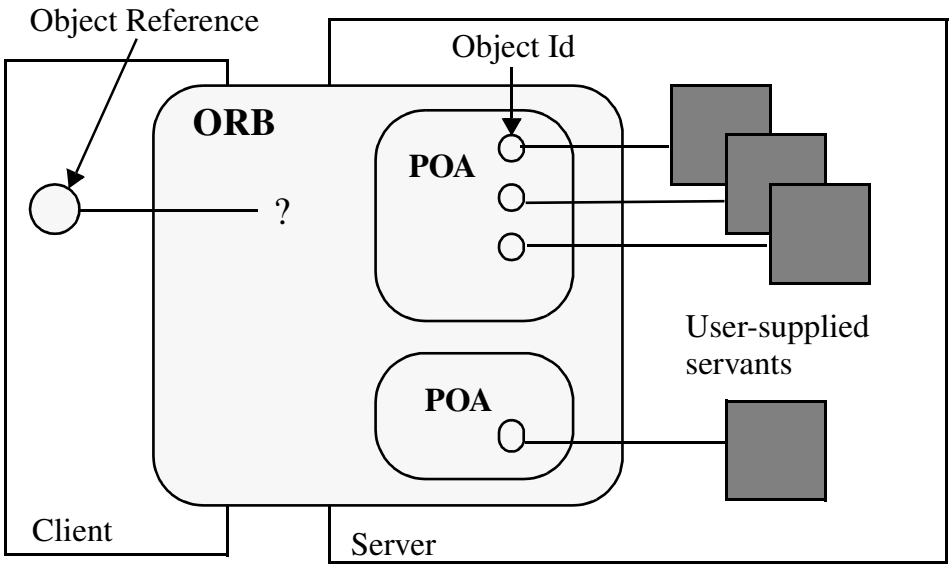

Figure 11-1 Abstract POA Model

If the POA supports the **RETAIN** policy, it maintains a map, labeled *Active Object Map*, that associates Object Ids with active servants, each association constituting an active object. If the POA has the **USE_DEFAULT_SERVANT** policy, a default servant may be registered with the POA. Alternatively, if the POA has the **USE_SERVANT_MANAGER** policy, a user-written servant manager may be registered with the POA. If the Active Object Map is not used, or a request arrives for an object not present in the Active Object Map, the POA either uses the default servant to perform the request or it invokes the servant manager to obtain a servant to perform the request. If the **RETAIN** policy is used, the servant returned by a servant manager is retained in the Active Object Map. Otherwise, the servant is used only to process the one request.

In this specification, the term *active* is applied equally to servants, Object Ids, and objects. An object is active in a POA if the POA's Active Object Map contains an entry that associates an Object Id with an existing servant. When this specification refers to *active Object Ids* and *active servants*, it means that the Object Id value or servant in question is part of an entry in the Active Object Map. An Object Id can appear in a POA's Active Object Map only once.

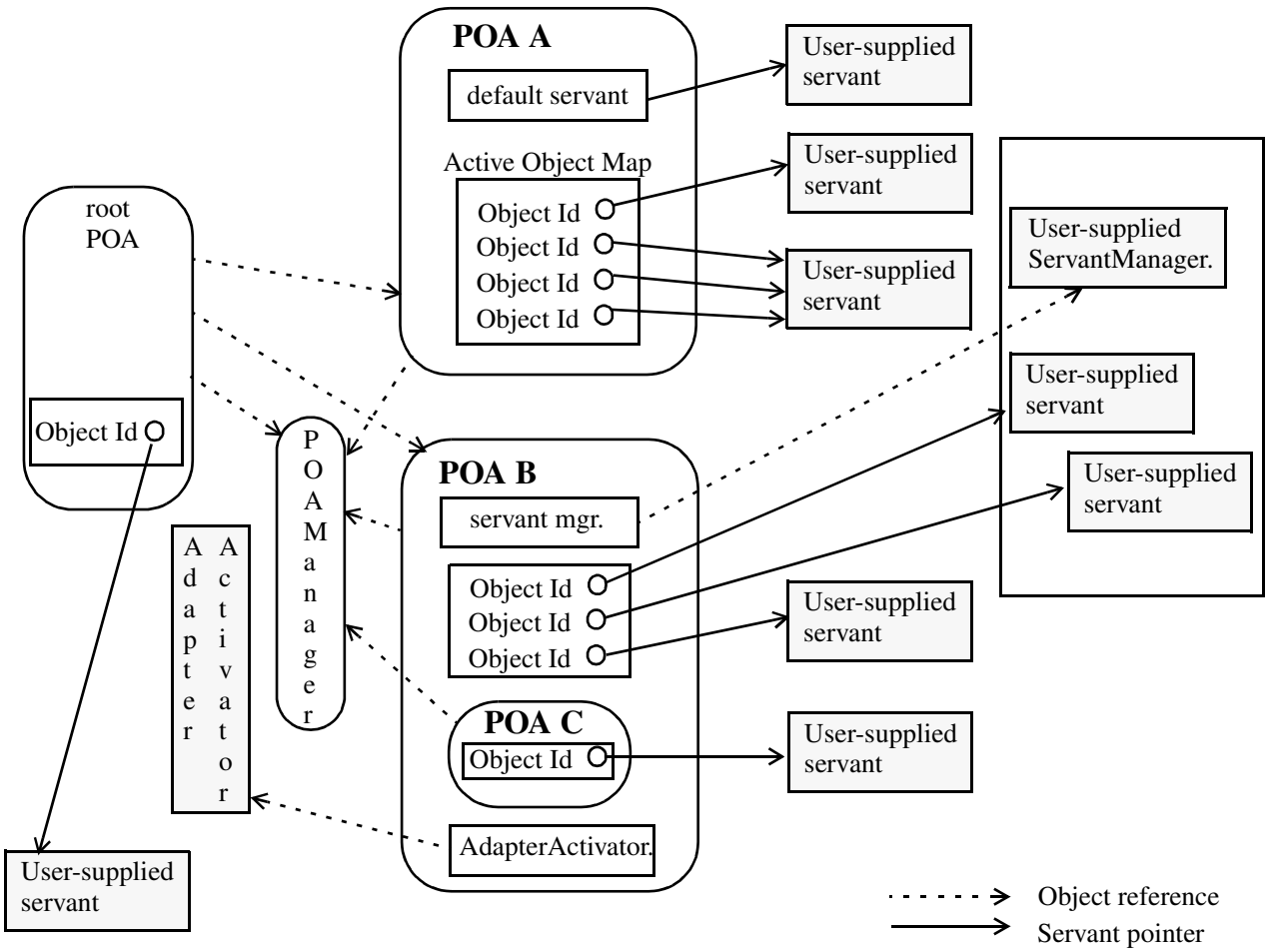

Figure 11-2 POA Architecture

11.2.3 POA Creation

To implement an object using the POA requires that the server application obtain a POA object. A distinguished POA object, called the *root POA*, is managed by the ORB and provided to the application using the ORB initialization interface under the initial object name “RootPOA.” The application developer can create objects using the root POA if those default policies are suitable. The root POA has the following policies.

- Thread Policy: **ORB_CTRL_MODEL**
- Lifespan Policy: **TRANSIENT**
- Object Id Uniqueness Policy: **UNIQUE_ID**
- Id Assignment Policy: **SYSTEM_ID**
- Servant Retention Policy: **RETAIN**
- Request Processing Policy: **USE_ACTIVE_OBJECT_MAP_ONLY**
- Implicit Activation Policy: **IMPLICIT_ACTIVATION**

The developer can also create new POAs. Creating a new POA allows the application developer to declare specific policy choices for the new POA and to provide a different adapter activator and servant manager (these are callback objects used by the POA to activate objects and nested POAs on demand). Creating new POAs also allows the application developer to partition the name space of objects, as Object Ids are interpreted relative to a POA. Finally, by creating new POAs, the developer can independently control request processing for multiple sets of objects.

A POA is created as a child of an existing POA using the **create_POA** operation on the parent POA. When a POA is created, the POA is given a name that must be unique with respect to all other POAs with the same parent.

POA objects are not persistent. No POA state can be assumed to be saved by the ORB. It is the responsibility of the server application to create and initialize the appropriate POA objects during server initialization or to set an AdapterActivater to create POA objects needed later.

Creating the appropriate POA objects is particularly important for persistent objects, objects whose existence can span multiple server lifetimes. To support an object reference created in a previous server process, the application must recreate the POA that created the object reference as well as all of its ancestor POAs. To ensure portability, each POA must be created with the same name as the corresponding POA in the original server process and with the same policies. (It is the user's responsibility to create the POA with these conditions.)

A portable server application can presume that there is no conflict between its POA names and the POA names chosen by other applications. It is the responsibility of the ORB implementation to provide a way to support this behavior.

11.2.4 Reference Creation

Object references are created in servers. Once they are created, they may be exported to clients.

From this model's perspective, object references encapsulate object identity information and information required by the ORB to identify and locate the server and POA with which the object is associated (that is, in whose scope the reference was created.) References are created in the following ways:

- The server application may directly create a reference with the **create_reference** and **create_reference_with_id** operations on a POA object. These operations collect the necessary information to constitute the reference, either from information associated with the POA or as parameters to the operation. These operations only create a reference. In doing so, they bring the abstract object into existence, but do not associate it with an active servant.
- The server application may explicitly activate a servant, associating it with an object identity using the **activate_object** or **activate_object_with_id** operations. Once a servant is activated, the server application can map the servant to its corresponding reference using the **servant_to_reference** or **id_to_reference** operations.

- The server application may cause a servant to implicitly activate itself. This behavior can only occur if the POA has been created with the **IMPLICIT_ACTIVATION** policy. If an attempt is made to obtain an object reference corresponding to an inactive servant, the POA may automatically assign a generated unique Object Id to the servant and activate the resulting object. The reference may be obtained by invoking **POA::servant_to_reference** with an inactive servant, or by performing an explicit or implicit type conversion from the servant to a reference type in programming language mappings that permit this conversion.

Once a reference is created in the server, it can be made available to clients in a variety of ways. It can be advertised through the OMG Naming and Trading Services. It can be converted to a string via **ORB::object_to_string** and published in some way that allows the client to discover the string and convert it to a reference using **ORB::string_to_object**. It can be returned as the result of an operation invocation.

Once a reference becomes available to a client, that reference constitutes the identity of the object from the client's perspective. As long as the client program holds and uses that reference, requests made on the reference should be sent to the "same" object.

Note – The meaning of object identity and "sameness" is at present the subject of debate in the OMG. This specification does not attempt to resolve that debate in any way, particularly by defining a concrete notion of identity that is exposed to clients, beyond the existing notions of identity described in the CORBA specifications and the OMA guide.

The states of servers and implementation objects are opaque to clients. This specification deals primarily with the view of the ORB from the server's perspective.

11.2.5 Object Activation States

At any point in time, a CORBA object may or may not be associated with an active servant.

If the POA has the **RETAIN** policy, the servant and its associated Object Id are entered into the Active Object Map of the appropriate POA. This type of activation can be accomplished in one of the following ways.

- The server application itself explicitly activates individual objects (via the **activate_object** or **activate_object_with_id** operations).
- The server application instructs the POA to activate objects on demand by having the POA invoke a user-supplied servant manager. The server application registers this servant manager with **set_servant_manager**.
- Under some circumstances (when the **IMPLICIT_ACTIVATION** policy is also in effect and the language binding allows such an operation), the POA may implicitly activate an object when the server application attempts to obtain a reference for a servant that is not already active (that is, not associated with an Object Id).

If the **USE_DEFAULT_SERVANT** policy is also in effect, the server application instructs the POA to activate unknown objects by having the POA invoke a single servant no matter what the Object Id is. The server application registers this servant with **set_servant**.

If the POA has the **NON_RETAIN** policy, for every request, the POA may use either a default servant or a servant manager to locate an active servant. From the POA's point of view, the servant is active only for the duration of that one request. The POA does not enter the servant-object association into the Active Object Map.

11.2.6 Request Processing

A request must be capable of conveying the Object Id of the target object as well as the identification of the POA that created the target object reference. When a client issues a request, the ORB first locates an appropriate server (perhaps starting one if needed) and then it locates the appropriate POA within that server.

If the POA does not exist in the server process, the application has the opportunity to re-create the required POA by using an adapter activator. An adapter activator is a user-implemented object that can be associated with a POA. It is invoked by the ORB when a request is received for a non-existent child POA. The adapter activator has the opportunity to create the required POA. If it does not, the client receives the OBJECT_NOT_EXIST exception.

Once the ORB has located the appropriate POA, it delivers the request to that POA. The further processing of that request depends both upon the policies associated with that POA as well as the object's current state of activation.

If the POA has the **RETAIN** policy, the POA looks in the Active Object Map to find out if there is a servant associated with the Object Id value from the request. If such a servant exists, the POA invokes the appropriate method on the servant.

If the POA has the **NON_RETAIN** policy or has the **RETAIN** policy but didn't find a servant in the Active Object Map, the POA takes the following actions:

- If the POA has the **USE_DEFAULT_SERVANT** policy, a default servant has been associated with the POA so the POA will invoke the appropriate method on that servant. If no servant has been associated with the POA, the POA raises the OBJ_ADAPTER system exception.
- If the POA has the **USE_SERVANT_MANAGER** policy, a servant manager has been associated with the POA so the POA will invoke **incarnate** or **preinvoke** on it to find a servant that may handle the request. (The choice of method depends on the **NON_RETAIN** or **RETAIN** policy of the POA.) If no servant manager has been associated with the POA, the POA raises the OBJ_ADAPTER system exception.
- If the **USE_OBJECT_MAP_ONLY** policy is in effect, the POA raises the OBJECT_NOT_EXIST system exception.

If a servant manager is located and invoked, but the servant manager is not directly capable of incarnating the object, it (the servant manager) may deal with the circumstance in a variety of ways, all of which are the application's responsibility.

Any system exception raised by the servant manager will be returned to the client in the reply. In addition to standard CORBA exceptions, a servant manager is capable of raising a ForwardRequest exception. This exception includes an object reference. The ORB will process this exception as stated below.

11.2.7 Implicit Activation

A POA can be created with a policy that indicates that its objects may be implicitly activated. This policy, **IMPLICIT_ACTIVATION**, also requires the **SYSTEM_ID** and **RETAIN** policies. When a POA supports implicit activation, an inactive servant may be implicitly activated in that POA by certain operations that logically require an Object Id to be assigned to that servant. Implicit activation of an object involves allocating a system-generated Object Id and registering the servant with that Object Id in the Active Object Map. The interface associated with the implicitly activated object is determined from the servant (using static information from the skeleton, or, in the case of a dynamic servant, using the **_primary_interface()** operation).

The operations that support implicit activation include:

- The **POA::servant_to_reference** operation, which takes a servant parameter and returns a reference.
- The **POA::servant_to_id** operation, which takes a servant parameter and returns an Object Id.
- Operations supported by a language mapping to obtain an object reference or an Object Id for a servant. For example, the **`_this()`** servant member function in C++ returns an object reference for the servant.
- Implicit conversions supported by a language mapping that convert a servant to an object reference or an Object Id.

The last two categories of operations are language-mapping-dependent.

If the POA has the **UNIQUE_ID** policy, then implicit activation will occur when any of these operations are performed on a servant that is not currently active (that is, it is associated with no Object Id in the POA's Active Object Map).

If the POA has the **MULTIPLE_ID** policy, the **servant_to_reference** and **servant_to_id** operations will *always* perform implicit activation, even if the servant is already associated with an Object Id. The behavior of language mapping operations in the **MULTIPLE_ID** case is specified by the language mapping. For example, in C++, the **`_this()`** servant member function will not implicitly activate a **MULTIPLE_ID** servant if the invocation of **`_this()`** is immediately within the dynamic context of a request invocation directed by the POA to that servant; instead, it returns the object reference used to issue the request.

Note – The exact timing of implicit activation is ORB implementation-dependent. For example, instead of activating the object immediately upon creation of a local object reference, the ORB could defer the activation until the Object Id is actually needed (for example, when the object reference is exported outside the process).

11.2.8 Multi-threading

The POA does not require the use of threads and does not specify what support is needed from a threads package. However, in order to allow the development of portable servers that utilize threads, the behavior of the POA and related interfaces when used within a multiple-thread environment must be specified.

Specifying this behavior does not require that an ORB must support being used in a threaded environment, nor does it require that an ORB must utilize threads in the processing of requests. The only requirement given here is that if an ORB does provide support for multi-threading, these are the behaviors that will be supported by that ORB. This allows a programmer to take advantage of multiple ORBs that support threads in a portable manner across those ORBs.

The POA's processing is affected by the thread-related calls available in the ORB: **work_pending**, **perform_work**, **run**, and **shutdown**.

11.2.8.1 POA Threading Models

The POA supports two models of threading when used in conjunction with multi-threaded ORB implementations; ORB controlled and single thread behavior. The two models can be used together or independently. Either model can be used in environments where a single-threaded ORB is used.

The threading model associated with a POA is indicated when the POA is created by including a **ThreadPolicy** object in the policies parameter of the POA's **create_POA** operation. Once a POA is created with one model, it cannot be changed to the other. All uses of the POA within the server must conform to that threading model associated with the POA.

11.2.8.2 Using the Single Thread Model

Requests for a single-threaded POA are processed sequentially. In a multi-threaded environment, all upcalls made by this POA to implementation code (servants, servant managers, and adapter activators) are made in a manner that is safe for code that is multi-thread-unaware.

11.2.8.3 Using the ORB Controlled Model

The ORB controlled model of threading is used in environments where the developer wants the ORB/POA to control the use of threads in the manner provided by the ORB. This model can also be used in environments that do not support threads.

In this model, the ORB is responsible for the creation, management, and destruction of threads used with one or more POAs.

11.2.8.4 Limitations When Using Multiple Threads

There are no guarantees that the ORB and POA will do anything specific about dispatching requests across threads with a single POA. Therefore, a server programmer who wants to use one or more POAs within multiple threads must take on all of the serialization of access to objects within those threads.

There may be requests active for the same object being dispatched within multiple threads at the same time. The programmer must be aware of this possibility and code with it in mind.

11.2.9 Dynamic Skeleton Interface

The POA is designed to enable programmers to connect servants to:

- type-specific skeletons, typically generated by OMG IDL compilers, or
- dynamic skeletons.

Servants that are members of type-specific skeleton classes are referred to as type-specific servants. Servants connected to dynamic skeletons are used to implement the Dynamic Skeleton Interface (DSI) and are referred to as DSI servants.

Whether a CORBA object is being incarnated by a DSI servant or a type-specific servant is transparent to its clients. Two CORBA objects supporting the same interface may be incarnated, one by a DSI servant and the other with a type-specific servant. Furthermore, a CORBA object may be incarnated by a DSI servant only during some period of time, while the rest of the time is incarnated by a static servant.

The mapping for POA DSI servants is language-specific, with each language providing a set of interfaces to the POA. These interfaces are used only by the POA. The interfaces required are the following.

- Take a **CORBA::ServerRequest** object from the POA and perform the processing necessary to execute the request.
- Return the Interface Repository Id identifying the most-derived interface supported by the target CORBA object in a request.

The reason for the first interface is the entire reason for existence of the DSI: to be able to handle any request in the way the programmer wishes to handle it. A single DSI servant may be used to incarnate several CORBA objects, potentially supporting different interfaces.

The reason for the second interface can be understood by comparing DSI servants to type-specific servants.

A type-specific servant may incarnate several CORBA objects but all of them will support the same IDL interface as the most-derived IDL interface. In C++, for example, an IDL interface **`Window`** in module **`GraphicalSystem`** will generate a type-specific skeleton class called **`Window`** in namespace **`POA_GraphicalSystem`**. A type-specific servant which is directly derived from the

POA_GraphicalSystem::Window skeleton class may incarnate several CORBA objects at a time, but all those CORBA objects will support the **GraphicalSystem::Window** interface as the most-derived interface.

A DSI servant may incarnate several CORBA objects, not necessarily supporting the same IDL interface as the most-derived IDL interface.

In both cases (type-specific and DSI) the POA may need to determine, at runtime, the Interface Repository Id identifying the most-derived interface supported by the target CORBA object in a request. The POA should be able to determine this by asking the servant that is going to serve the CORBA object.

In the case of type-specific servants, the POA obtains that information from the type-specific skeleton class from which the servant is directly derived. In the case of DSI servants, the POA obtains that information by using the second language-specific interface above.

11.2.10 Location Transparency

The POA supports location transparency for objects implemented using the POA. Unless explicitly stated to the contrary, all POA behavior described in this specification applies regardless of whether the client is local (same process) or remote. For example, like a request from a remote client, a request from a local client may cause object activation if the object is not active, block indefinitely if the target object's POA is in the holding state, be rejected if the target object's POA is in the discarding or inactive states, be delivered to a thread-unaware object implementation, or be delivered to a different object if the target object's servant manager raises the ForwardRequest exception. The Object Id and POA of the target object will also be available to the server via the **Current** object, regardless of whether the client is local or remote.

Note – The implication of these requirements on the ORB implementation is to require the ORB to mediate all requests to POA-based objects, even if the client is co-resident in the same process. This specification is not intended to change CORBAServices specifications that allow for behaviors that are not location transparent. This specification does not prohibit (nonstandard) POA extensions to support object behavior that is not location-transparent.

11.3 Interfaces

The POA-related interfaces are defined in a module separate from the **CORBA** module, the **PortableServer** module. It consists of these interfaces:

- **POA**
- **POAManager**
- **ServantManager**
- **ServantActivator**
- **ServantLocator**

- **AdapterActivator**
- **ThreadPolicy**
- **LifespanPolicy**
- **IdUniquenessPolicy**
- **IdAssignmentPolicy**
- **ImplicitActivationPolicy**
- **ServantRetentionPolicy**
- **RequestProcessingPolicy**
- **Current**

In addition, the POA defines the **Servant** native type.

11.3.1 The Servant IDL Type

This specification defines a native type **PortableServer::Servant**. Values of the type **Servant** are programming-language-specific implementations of CORBA interfaces. Each language mapping must specify how **Servant** is mapped to the programming language data type that corresponds to an object implementation. The **Servant** type has the following characteristics and constraints.

- Values of type **Servant** are opaque from the perspective of CORBA application programmers. There are no operations that can be performed directly on them by user programs. They can be passed as parameters to certain POA operations. Some language mappings may allow **Servant** values to be implicitly converted to object references under appropriate conditions.
- Values of type **Servant** support a language-specific programming interface that can be used by the ORB to obtain a default POA for that servant. This interface is used only to support implicit activation. A language mapping may provide a default implementation of this interface that returns the root POA of a default ORB.
- Values of type Servant provide default implementations of the standard object reference operations **get_interface**, **is_a**, and **non_existent**. These operations can be overridden by the programmer to provide additional behavior needed by the object implementation. The default implementations of **get_interface** and **is_a** operations use the most derived interface of a static servant or the most derived interface retrieved from a dynamic servant to perform the operation. The default implementation of the **non_existent** operation returns **FALSE**. These operations are invoked by the POA just like any other operation invocation, so the **PortableServer::Current** interface and any language-mapping-provided method of accessing the invocation context are available.
- Values of type **Servant** must be testable for identity.
- Values of type **Servant** have no meaning outside of the process context or address space in which they are generated.

11.3.2 POAManager Interface

Each POA object has an associated **POAManager** object. A POA manager may be associated with one or more POA objects. A POA manager encapsulates the processing state of the POAs it is associated with. Using operations on the POA manager, an application can cause requests for those POAs to be queued or discarded, and can cause the POAs to be deactivated.

POA managers are created and destroyed implicitly. Unless an explicit POA manager object is provided at POA creation time, a POA manager is created when a POA is created and is automatically associated with that POA. A POA manager object is implicitly destroyed when all of its associated POAs have been destroyed.

11.3.2.1 Processing States

A POA manager has four possible processing states; *active*, *inactive*, *holding*, and *discarding*. The processing state determines the capabilities of the associated POAs and the disposition of requests received by those POAs. Figure 11-3 on page 11-16 illustrates the processing states and the transitions between them. For simplicity of presentation, this specification sometimes describes these states as POA states, referring to the POA or POAs that have been associated with a particular POA manager. A POA manager is created in the *holding* state. The root POA is therefore initially in the *holding* state.

For simplicity in the figure and the explanation, operations that would not cause a state change are not shown. For example, if a POA is in “active” state, it does not change state due to an activate operation. Such operations complete successfully with no special notice.

The only exception is the inactive state: a “deactivate” operation raises an exception just the same as every other attempted state change operation.

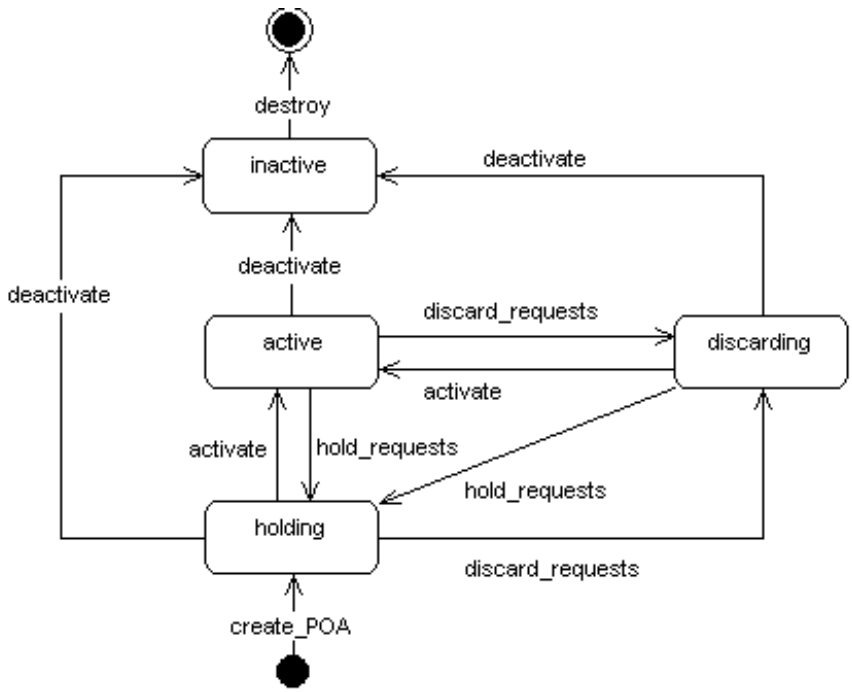

Figure 11-3 Processing States

Active State

When a POA manager is in the *active* state, the associated POAs will receive and start processing requests (assuming that appropriate thread resources are available). Note that even in the active state, a POA may need to queue requests depending upon the ORB implementation and resource limits. The number of requests that can be received and/or queued is an implementation limit. If this limit is reached, the POA should return a TRANSIENT system exception to indicate that the client should re-issue the request.

A user program can legally transition a POA manager from the *active* state to either the *discarding*, *holding*, or *inactive* state by calling the **discard_requests**, **hold_requests**, or **deactivate** operations, respectively. The POA enters the *active* state through the use of the **activate** operation when in the *discarding* or *holding* state.

Discarding State

When a POA manager is in the *discarding* state, the associated POAs will discard all incoming requests (whose processing has not yet begun). When a request is discarded, the TRANSIENT system exception must be returned to the client-side to indicate that the request should be re-issued. (Of course, an ORB may always reject a request for other reasons and raise some other system exception.)

In addition, when a POA manager is in the *discarding* state, the adapter activators registered with the associated POAs will not get called. Instead, requests that require the invocation of an adapter activator will be discarded, as described in the previous paragraph.

The primary purpose of the *discarding* state is to provide an application with flow-control capabilities when it determines that an object's implementation or POA is being flooded with requests. It is expected that the application will restore the POA manager to the *active* state after correcting the problem that caused flow-control to be needed.

A POA manager can legally transition from the *discarding* state to either the *active*, *holding*, or *inactive* state by calling the **activate**, **hold_requests**, or **deactivate** operations, respectively. The POA enters the *discarding* state through the use of the **discard_requests** operation when in the *active* or *holding* state.

Holding State

When a POA manager is in the *holding* state, the associated POAs will queue incoming requests. The number of requests that can be queued is an implementation limit. If this limit is reached, the POAs may discard requests and return the TRANSIENT system exception to the client to indicate that the client should reissue the request. (Of course, an ORB may always reject a request for other reasons and raise some other system exception.)

In addition, when a POA manager is in the *holding* state, the adapter activators registered with the associated POAs will not get called. Instead, requests that require the invocation of an adapter activator will be queued, as described in the previous paragraph.

A POA manager can legally transition from the *holding* state to either the *active*, *discarding*, or *inactive* state by calling the **activate**, **discard_requests**, or **deactivate** operations, respectively. The POA enters the *holding* state through the use of the **hold_requests** operation when in the *active* or *discarding* state. A POA manager is created in the holding state.

Inactive State

The *inactive* state is entered when the associated POAs are to be shut down. Unlike the *discarding* state, the *inactive* state is not a temporary state. When a POA manager is in the *inactive* state, the associated POAs will reject new requests. The rejection mechanism used is specific to the vendor. The GIOP location forwarding mechanism and CloseConnection message are examples of mechanisms that could be used to indicate the rejection. If the client is co-resident in the same process, the ORB could raise the OBJ_ADAPTER exception to indicate that the object implementation is unavailable.

In addition, when a POA manager is in the *inactive* state, the adapter activators registered with the associated POAs will not get called. Instead, requests that require the invocation of an adapter activator will be rejected, as described in the previous paragraph.

The *inactive* state is entered using the **deactivate** operation. It is legal to enter the *inactive* state from either the *active*, *holding*, or *discarding* states.

If the transition into the *inactive* state is a result of calling **deactivate** with an **etherealize_objects** parameter of

- TRUE - the associated POAs will call **etherealize** for each active object associated with the POA once all currently executing requests have completed processing (if the POAs have the **RETAIN** and **USE_SERVANT_MANAGER** policies). If a servant manager has been registered for the POA, the POA will get rid of the object. If there are any queued requests that have not yet started executing, they will be treated as if they were new requests and rejected.
- FALSE - No deactivations or etherealizations will be attempted.

11.3.2.2 Locality Constraints

A **POAManager** object must not be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a MARSHAL system exception. An attempt to use a **POAManager** object with the DII may raise the NO_IMPLEMENT exception.

11.3.2.3 activate

```
void activate()
    raises (AdapterInactive);
```

This operation changes the state of the POA manager to *active*. If issued while the POA manager is in the *inactive* state, the AdapterInactive exception is raised. Entering the *active* state enables the associated POAs to process requests.

11.3.2.4 hold_requests

```
void hold_requests( in boolean wait_for_completion )
    raises(AdapterInactive);
```

This operation changes the state of the POA manager to *holding*. If issued while the POA manager is in the *inactive* state, the AdapterInactive exception is raised. Entering the *holding* state causes the associated POAs to queue incoming requests. Any requests that have been queued but have not started executing will continue to be queued while in the *holding* state.

If the **wait_for_completion** parameter is **FALSE**, this operation returns immediately after changing the state. If the parameter is **TRUE** and the current thread is not in an invocation context dispatched by some POA belonging to the same ORB as this POA, this operation does not return until either there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) or the state of the POA manager is changed to a state other than *holding*. If the parameter is **TRUE** and the current thread is in an invocation context dispatched by some POA belonging to the same ORB as this POA the BAD_INV_ORDER exception is raised and the state is not changed.

11.3.2.5 discard_requests

void discard_requests(in boolean wait_for_completion)
raises (AdapterInactive);

This operation changes the state of the POA manager to *discarding*. If issued while the POA manager is in the *inactive* state, the AdapterInactive exception is raised. Entering the *discarding* state causes the associated POAs to discard incoming requests. In addition, any requests that have been queued but have not started executing are discarded. When a request is discarded, a TRANSIENT system exception is returned to the client.

If the **wait_for_completion** parameter is FALSE, this operation returns immediately after changing the state. If the parameter is TRUE and the current thread is not in an invocation context dispatched by some POA belonging to the same ORB as this POA, this operation does not return until either there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) or the state of the POA manager is changed to a state other than *discarding*. If the parameter is TRUE and the current thread is in an invocation context dispatched by some POA belonging to the same ORB as this POA the BAD_INV_ORDER exception is raised and the state is not changed.

11.3.2.6 deactivate

void deactivate(in boolean etherealize_objects,
in boolean wait_for_completion);
raises (AdapterInactive);

This operation changes the state of the POA manager to *inactive*. If issued while the POA manager is in the *inactive* state, the AdapterInactive exception is raised. Entering the inactive state causes the associated POAs to reject requests that have not begun to be executed as well as any new requests.

After changing the state, if the **etherealize_objects** parameter is

- TRUE - the POA manager will cause all associated POAs that have the **RETAIN** and **USE_SERVANT_MANAGER** policies to perform the **etherealize** operation on the associated servant manager for all active objects.
- FALSE - the **etherealize** operation is not called. The purpose is to provide developers with a means to shut down POAs in a crisis (for example, unrecoverable error) situation.

If the **wait_for_completion** parameter is FALSE, this operation will return immediately after changing the state. If the parameter is TRUE and the current thread is not in an invocation context dispatched by some POA belonging to the same ORB as this POA, this operation does not return until there are no actively executing requests in any of the POAs associated with this POA manager (that is, all requests that were started prior to the state change have completed) and, in the case of a TRUE **etherealize_objects**, all invocations of **etherealize** have completed for POAs having the **RETAIN** and **USE_SERVANT_MANAGER** policies. If the parameter is **TRUE**

and the current thread is in an invocation context dispatched by some POA belonging to the same ORB as this POA the BAD_INV_ORDER exception is raised and the state is not changed.

If the **ORB::shutdown** operation is called, it makes a call on **deactivate** with a TRUE **etherealize_objects** parameter for each POA manager known in the process; the **wait_for_completion** parameter to **deactivate** will be the same as the similarly named parameter of **ORB::shutdown**.

If **deactivate** is called multiple times before destruction is complete (because there are active requests), the **etherealize_objects** parameter applies only to the first call of **deactivate;** subsequent calls with conflicting **etherealize_objects** settings will use the value of the **etherealize_objects** from the first call. The **wait_for_completion** parameter will be handled as defined above for each individual call (some callers may choose to block, while others may not).

11.3.2.7 *get_state*

```
enum State {HOLDING, ACTIVE, DISCARDING, INACTIVE};
State get_state();
```

This operation returns the state of the POA manager.

11.3.3 *AdapterActivator Interface*

Adapter activators are associated with POAs. An adapter activator supplies a POA with the ability to create child POAs on demand, as a side-effect of receiving a request that names the child POA (or one of its children), or when **find_POA** is called with an activate parameter value of TRUE. An application server that creates all its needed POAs at the beginning of execution does not need to use or provide an adapter activator; it is necessary only for the case in which POAs need to be created during request processing.

While a request from the POA to an adapter activator is in progress, all requests to objects managed by the new POA (or any descendant POAs) will be queued. This serialization allows the adapter activator to complete any initialization of the new POA before requests are delivered to that POA.

11.3.3.1 *Locality Constraints*

An AdapterActivator object must be local to the process containing the POA objects it is registered with.

11.3.3.2 *unknown_adapter*

```
boolean unknown_adapter(in POA parent, in string name);
```

This operation is invoked when the ORB receives a request for an object reference that identifies a target POA that does not exist. The ORB invokes this operation once for each POA that must be created in order for the target POA to exist (starting with the

ancestor POA closest to the root POA). The operation is invoked on the adapter activator associated with the POA that is the parent of the POA that needs to be created. That parent POA is passed as the **parent** parameter. The name of the POA to be created (relative to the parent) is passed as the **name** parameter.

The implementation of this operation should either create the specified POA and return TRUE, or it should return FALSE. If the operation returns TRUE, the ORB will proceed with processing the request. If the operation returns FALSE, the ORB will return OBJECT_NOT_EXIST to the client. If multiple POAs need to be created, the ORB will invoke **unknown_adapter** once for each POA that needs to be created. If the parent of a nonexistent POA does not have an associated adapter activator, the ORB will return the OBJECT_NOT_EXIST exception.

If **unknown_adapter** raises a system exception, the ORB will report an OBJ_ADAPTER exception.

Note – It is possible for another thread to create the same POA the AdapterActivator is being asked to create if AdapterActivators are used in conjunction with other threads calling **create_POA** with the same POA name. Applications should be prepared to deal with failures from either the manual or automatic (AdapterActivator) POA creation request. There can be no guarantee of the order of such calls.

For example, if the target object reference was created by a POA whose full name is “A”, “B”, “C”, “D” and only POAs “A” and “B” currently exist, the **unknown_adapter** operation will be invoked on the adapter activator associated with POA “B” passing POA “B” as the parent parameter and “C” as the name of the missing POA. Assuming that the adapter activator creates POA “C” and returns TRUE, the ORB will then invoke **unknown_adapter** on the adapter activator associated with POA “C”, passing POA “C” as the parent parameter and “D” as the name.

The **unknown_adapter** operation is also invoked when **find_POA** is called on the POA with which the AdapterActivator is associated, the specified child does not exist, and the **activate_it** parameter to **find_POA** is TRUE. If **unknown_adapter** creates the specified POA and returns TRUE, that POA is returned from **find_POA**.

Note – This allows the same code, the **unknown_adapter** implementation, to be used to initialize a POA whether that POA is created explicitly by the application or as a side-effect of processing a request. Furthermore, it makes this initialization atomic with respect to delivery of requests to the POA.

11.3.4 ServantManager Interface

Servant managers are associated with POAs. A servant manager supplies a POA with the ability to activate objects on demand when the POA receives a request targeted at an inactive object. A servant manager is registered with a POA as a callback object, to be invoked by the POA when necessary. An application server that activates all its needed objects at the beginning of execution does not need to use a servant manager; it is used only for the case in which an object must be activated during request processing.

The **ServantManager** interface is itself empty. It is inherited by two other interfaces, **ServantActivator** and **ServantLocator**.

The two types of servant managers correspond to the POA's **RETAIN** policy (**ServantActivator**) and to the **NON_RETAIN** policy (**ServantLocator**). The meaning of the policies and the operations that are available for POAs using each policy are listed under the two types of derived interfaces.

Each servant manager type contains two operations, the first called to find and return a servant and the second to deactivate a servant. The operations differ according to the amount of information usable for their situation.

11.3.4.1 Common information for servant manager types

The two types of servant managers have certain semantics that are identical.

The **incarnate** and **preinvoke** operation may raise any system exception deemed appropriate (for example, OBJECT_NOT_EXIST if the object corresponding to the Object Id value has been destroyed).

Note – If a user-written routine (servant manager or method code) raises the OBJECT_NOT_EXIST exception, the POA does nothing but pass on that exception. It is the user's responsibility to deactivate the object if it had been previously activated.

The **incarnate** and **preinvoke** operation may also raise a ForwardRequest exception. If this occurs, the ORB is responsible for delivering the current request and subsequent requests to the object denoted in the **forward_reference** member of the exception. The behavior of this mechanism must be the functional equivalent of the GIOP location forwarding mechanism. If the current request was delivered via an implementation of the GIOP protocol (such as IIOP), the reference in the exception should be returned to the client in a reply message with **LOCATION_FORWARD** reply status. If some other protocol or delivery mechanism was used, the ORB is responsible for providing equivalent behavior, from the perspectives of the client and the object denoted by the new reference.

If a **ServantManager** returns a null Servant (or the equivalent in a language mapping) as the result of an **incarnate()** or **preinvoke()** operation, the POA will return the OBJ_ADAPTER system exception as the result of the request. If the **ServantManager** returns the wrong type of Servant, it is indeterminate when that error is detected. It is likely to result in a BAD_OPERATION or MARSHAL exception at the time of method invocation.

11.3.4.2 Locality Constraints

A ServantManager object must be local to the process containing the POA objects it is registered with.

11.3.5 ServantActivator Interface

When the POA has the **RETAIN** policy it uses servant managers that are ServantActivators. When using such servant managers, the following statements apply for a given **ObjectId** used in the **incarnate** and **etherealize** operations:

- Servants incarnated by the servant manager will be placed in the Active Object Map with objects they have activated.
- Invocations of **incarnate** on the servant manager are serialized.
- Invocations of **etherealize** on the servant manager are serialized.
- Invocations of **incarnate** and **etherealize** on the servant manager are mutually exclusive.
- Incarnations of a particular object may not overlap; that is, **incarnate** shall not be invoked with a particular **ObjectId** while, within the same POA, that **ObjectId** is in use as the **ObjectId** of an activated object or as the argument of a call to incarnate or etherealize that has not completed.

It should be noted that there may be a period of time between an object's deactivation and the etherealization (during which outstanding requests are being processed) in which arriving requests on that object should not be passed to its servant. During this period, requests targeted for such an object act as if the POA were in *holding* state until **etherealize** completes. If **etherealize** is called as a consequence of a **deactivate** call with an **etherealize_objects** parameter of TRUE, incoming requests are rejected.

It should also be noted that a similar situation occurs with **incarnate**. There may be a period of time after the POA invokes **incarnate** and before that method returns in which arriving requests bound for that object should not be passed to the servant.

A single servant manager object may be concurrently registered with multiple POAs. Invocations of **incarnate** and **etherealize** on a servant manager in the context of different POAs are not necessarily serialized or mutually exclusive. There are no assumptions made about the thread in which **etherealize** is invoked.

11.3.5.1 incarnate

```
Servant incarnate (
    in ObjectId         oid,
    in POA              adapter)
    raises (ForwardRequest);
```

This operation is invoked by the POA whenever the POA receives a request for an object that is not currently active, assuming the POA has the **USE_SERVANT_MANAGER** and **RETAIN** policies.

The **oid** parameter contains the **ObjectId** value associated with the incoming request. The **adapter** is an object reference for the POA in which the object is being activated.

The user-supplied servant manager implementation is responsible for locating or creating an appropriate servant that corresponds to the **ObjectId** value if possible. **incarnate** returns a value of type Servant, which is the servant that will be used to process the incoming request (and potentially subsequent requests, since the POA has the **RETAIN** policy).

The POA enters the returned Servant value into the Active Object Map so that subsequent requests with the same **ObjectId** value will be delivered directly to that servant without invoking the servant manager.

If the **incarnate** operation returns a servant that is already active for a different Object Id and if the POA also has the **UNIQUE_ID** policy, the **incarnate** has violated the POA policy and is considered to be in error. The POA will raise an OBJ_ADAPTER system exception for the request.

Note – If the same servant is used in two different POAs, it is legal for the POAs to use that servant even if the POAs have different Object Id uniqueness policies. The POAs do not interact with each other in this regard.

11.3.5.2 etherealize

```
void etherealize (
    in ObjectId         oid,
    in POA              adapter,
    in Servant          serv,
    in boolean          cleanup_in_progress,
    in boolean          remaining_activations);
```

This operation is invoked whenever a servant for an object is deactivated, assuming the POA has the **USE_SERVANT_MANAGER** and **RETAIN** policies. Note that an active servant may be deactivated by the servant manager via **etherealize** even if it was not incarnated by the servant manager.

The **oid** parameter contains the Object Id value of the object being deactivated. The **adapter** parameter is an object reference for the **POA** in whose scope the object was active. The **serv** parameter contains a reference to the servant that is associated with the object being deactivated. If the servant denoted by the **serv** parameter is associated with other objects in the **POA** denoted by the **adapter** parameter (that is, in the **POA**'s Active Object Map) at the time that **etherealize** is called, the **remaining_activations** parameter has the value **TRUE**. Otherwise, it has the value **FALSE**.

If the **cleanup_in_progress** parameter is **TRUE**, the reason for the **etherealize** operation is that either the **deactivate** or **destroy** operation was called with an **etherealize_objects** parameter of **TRUE**. If the parameter is **FALSE**, the **etherealize** operation is called for other reasons.

Deactivation occurs in the following circumstances:

- When an object is deactivated explicitly by an invocation of **POA::deactivate_object**.
- When the ORB or POA determines internally that an object must be deactivated. For example, an ORB implementation may provide policies that allow objects to be deactivated after some period of quiescence, or when the number of active objects reaches some limit.
- If **POAManager::deactivate** is invoked on a POA manager associated with a POA that has currently active objects.

Destroying a servant that is in the Active Object Map or is otherwise known to the POA can lead to undefined results.

In a multi-threaded environment, the **POA** makes certain guarantees that allow servant managers to safely destroy servants. Specifically, the servant's entry in the Active Object Map corresponding to the target object is removed before **etherealize** is called. Because calls to **incarnate** and **etherealize** are serialized, this prevents new requests for the target object from being invoked on the servant during etherealization. After removing the entry from the Active Object Map, if the **POA** determines before invoking **etherealize** that other requests for the same target object are already in progress on the servant, it delays the call to **etherealize** until all active methods for the target object have completed. Therefore, when **etherealize** is called, the servant manager can safely destroy the servant if it wants to, unless the **remaining_activations** argument is TRUE.

If the **etherealize** operation returns a system exception, the **POA** ignores the exception.

11.3.6 ServantLocator Interface

When the POA has the **NON_RETAIN** policy it uses servant managers that are **ServantLocators**. Because the POA knows that the servant returned by this servant manager will be used only for a single request, it can supply extra information to the servant manager's operations and the servant manager's pair of operations may be able to cooperate to do something different than a ServantActivator.

When the POA uses the **ServantLocator** interface, immediately after performing the operation invocation on the servant returned by **preinvoke**, the POA will invoke **postinvoke** on the servant manager, passing the **ObjectId** value and the **Servant** value as parameters (among others). The next request with this **ObjectId** value will then cause **preinvoke** to be invoked again. This feature may be used to force every request for objects associated with a **POA** to be mediated by the servant manager.

When using such a **ServantLocator**, the following statements apply for a given **ObjectId** used in the **preinvoke** and **postinvoke** operations:

- The servant returned by **preinvoke** is used only to process the single request that caused **preinvoke** to be invoked.
- No servant incarnated by the servant manager will be placed in the Active Object Map.

- When the invocation of the request on the servant is complete, **postinvoke** will be invoked for the object.
- No serialization of invocations of **preinvoke** or **postinvoke** may be assumed; there may be multiple concurrent invocations of **preinvoke** for the same **ObjectId**. (However, if the **SINGLE_THREAD_MODEL** policy is being used, that policy will serialize these calls.)
- The same thread will be used to **preinvoke** the object, process the request, and **postinvoke** the object.
- The **preinvoke** and postinvoke operations are always called in pairs in response to any ORB activity. In particular, for a response to a **GIOP Locate** message a **GIOP**-conforming ORB may (or may not) call preinvoke to determine whether the object could be served at this location. If the ORB makes such a call, whatever the result, the ORB does not invoke a method, but does call postinvoke before responding to the **Locate** message. (Note that the **ServantActivator** interface does not behave similarly with respect to a **GIOP Locate** message since the etherealize operation is not associated with request processing.)

11.3.6.1 *preinvoke*

```
Servant preinvoke(
    in ObjectId             oid,
    in POA                  adapter,
    in CORBA::Identifier    operation,
    out Cookie              the_cookie)
    raises (ForwardRequest);
```

This operation is invoked by the POA whenever the POA receives a request for an object that is not currently active, assuming the POA has the **USE_SERVANT_MANAGER** and **NON_RETAIN** policies.

The **oid** parameter contains the **ObjectId** value associated with the incoming request. The **adapter** is an object reference for the POA in which the object is being activated.

The user-supplied servant manager implementation is responsible for locating or creating an appropriate servant that corresponds to the **ObjectId** value if possible. **preinvoke** returns a value of type **Servant**, which is the servant that will be used to process the incoming request.

The **Cookie** is a type opaque to the **POA** that can be set by the servant manager for use later by **postinvoke**. The operation is the name of the operation that will be called by the **POA** when the servant is returned.

11.3.6.2 *postinvoke*

```
void postinvoke(
    in ObjectId             oid,
    in POA                  adapter,
    in CORBA::Identifier    operation,
    in Cookie               the_cookie,
```

in Servant the_servant);

This operation is invoked whenever a servant completes a request, assuming the POA has the **USE_SERVANT_MANAGER** and **NON_RETAIN** policies.

The **postinvoke** operation is considered to be part of a request on an object. That is, the request is not complete until postinvoke finishes. If the method finishes normally but postinvoke raises a system exception, the method's normal return is overridden; the request completes with the exception.

The **oid** parameter contains the Object Id value of the object on which the request was made. The **adapter** parameter is an object reference for the POA in whose scope the object was active. The **the_servant** parameter contains a reference to the servant that is associated with the object.

The **Cookie** is a type opaque to the **POA**; it contains any value that was set by the **preinvoke** operation. The operation is the name of the operation that was called by the **POA** for the request.

Destroying a servant that is known to the **POA** can lead to undefined results.

11.3.7 POA Policy Objects

Interfaces derived from **CORBA::Policy** are used with the **POA::create_POA** operation to specify policies that apply to a POA. Policy objects are created using factory operations on any pre-existing POA, such as the root POA. Policy objects are specified when a POA is created. Policies may not be changed on an existing POA. Policies are not inherited from the parent POA.

11.3.7.1 Thread Policy

Objects with the **ThreadPolicy** interface are obtained using the **POA::create_thread_policy** operation and passed to the **POA::create_POA** operation to specify the threading model used with the created POA. The value attribute of **ThreadPolicy** contains the value supplied to the **POA::create_thread_policy** operation from which it was obtained. The following values can be supplied.

- **ORB_CTRL_MODEL** - The ORB is responsible for assigning requests for an ORB-controlled POA to threads. In a multi-threaded environment, concurrent requests may be delivered using multiple threads.
- **SINGLE_THREAD_MODEL** - Requests for a single-threaded POA are processed sequentially. In a multi-threaded environment, all upcalls made by this POA to implementation code (servants and servant managers) are made in a manner that is safe for code that is multi-thread-unaware.

If no **ThreadPolicy** object is passed to **create_POA**, the thread policy defaults to **ORB_CTRL_MODEL**.

Note – In some environments, calling multi-thread-unaware code safely (that is, using the **SINGLE_THREAD_MODEL)** may mean that the POA will use only the main thread, in which case the application programmer is responsible to ensure that the main thread is given to the ORB, using **ORB::perform_work** or **ORB::run**.

POAs using the **SINGLE_THREAD_MODEL** may need to cooperate to ensure that calls are safe even when implementation code (such as a servant manager) is shared by multiple single-threaded POAs.

These models presume that the ORB and the application are using compatible threading primitives in a multi-threaded environment.

11.3.7.2 Lifespan Policy

Objects with the **LifespanPolicy** interface are obtained using the **POA::create_lifespan_policy** operation and passed to the **POA::create_POA** operation to specify the lifespan of the objects implemented in the created POA. The following values can be supplied.

- **TRANSIENT** - The objects implemented in the **POA** cannot outlive the **POA** instance in which they are first created. Once the **POA** is deactivated, use of any object references generated from it will result in an OBJECT_NOT_EXIST exception.
- **PERSISTENT** - The objects implemented in the **POA** can outlive the process in which they are first created.
  - Persistent objects have a **POA** associated with them (the **POA** which created them). When the ORB receives a request on a persistent object, it first searches for the matching **POA**, based on the names of the **POA** and all of its ancestors.
  - Administrative action beyond the scope of this specification may be necessary to inform the ORB's location service of the creation and eventual termination of existence of this **POA**, and optionally to arrange for on-demand activation of a process implementing this **POA**.
  - **POA** names must be unique within their enclosing scope (the parent **POA**). A portable program can assume that **POA** names used in other processes will not conflict with its own **POA** names. A conforming CORBA implementation will provide a method for ensuring this property.

If no **LifespanPolicy** object is passed to **create_POA**, the lifespan policy defaults to **TRANSIENT**.

11.3.7.3 Object Id Uniqueness Policy

Objects with the **IdUniquenessPolicy** interface are obtained using the **POA::create_id_uniqueness_policy** operation and passed to the **POA::create_POA** operation to specify whether the servants activated in the created **POA** must have unique object identities. The following values can be supplied.

- **UNIQUE_ID** - Servants activated with that POA support exactly one Object Id.
- **MULTIPLE_ID** - a servant activated with that POA may support one or more Object Ids.

If no **IdUniquenessPolicy** is specified at POA creation, the default is **UNIQUE_ID**.

11.3.7.4 Id Assignment Policy

Objects with the **IdAssignmentPolicy** interface are obtained using the **POA::create_id_assignment_policy** operation and passed to the **POA::create_POA** operation to specify whether Object Ids in the created **POA** are generated by the application or by the ORB. The following values can be supplied.

- **USER_ID** - Objects created with that **POA** are assigned Object Ids only by the application.
- **SYSTEM_ID** - Objects created with that **POA** are assigned Object Ids only by the **POA**. If the **POA** also has the **PERSISTENT** policy, assigned Object Ids must be unique across all instantiations of the same POA.

If no **IdAssignmentPolicy** is specified at POA creation, the default is **SYSTEM_ID**.

11.3.7.5 Servant Retention Policy

Objects with the **ServantRetentionPolicy** interface are obtained using the **POA::create_servant_retention_policy** operation and passed to the **POA::create_POA** operation to specify whether the created **POA** retains active servants in an Active Object Map. The following values can be supplied.

- **RETAIN -** The POA will retain active servants in its Active Object Map.
- **NON_RETAIN -** Servants are not retained by the POA.

If no **ServantRetentionPolicy** is specified at POA creation, the default is **RETAIN**.

Note – The **NON_RETAIN** policy requires either the **USE_DEFAULT_SERVANT** or **USE_SERVANT_MANAGER** policies.

11.3.7.6 Request Processing Policy

Objects with the RequestProcessingPolicy interface are obtained using the **POA::create_request_processing_policy** operation and passed to the **POA::create_POA** operation to specify how requests are processed by the created **POA**. The following values can be supplied.

- **USE_ACTIVE_OBJECT_MAP_ONLY** - If the Object Id is not found in the Active Object Map, an OBJECT_NOT_EXIST exception is returned to the client. The **RETAIN** policy is also required.

- **USE_DEFAULT_SERVANT** - If the Object Id is not found in the Active Object Map or the **NON_RETAIN** policy is present, and a default servant has been registered with the **POA** using the **set_servant** operation, the request is dispatched to the default servant. If no default servant has been registered, an OBJ_ADAPTER exception is returned to the client. The **MULTIPLE_ID** policy is also required.
- **USE_SERVANT_MANAGER** - If the Object Id is not found in the Active Object Map or the **NON_RETAIN** policy is present, and a servant manager has been registered with the **POA** using the **set_servant_manager** operation, the servant manager is given the opportunity to locate a servant or raise an exception. If no servant manager has been registered, an OBJECT_ADAPTER exception is returned to the client.

If no **RequestProcessingPolicy** is specified at **POA** creation, the default is **USE_ACTIVE_OBJECT_MAP_ONLY**.

By means of combining the **USE_ACTIVE_OBJECT_MAP_ONLY** / **USE_DEFAULT_SERVANT** / **USE_SERVANT_MANAGER** policies and the **RETAIN** / **NON_RETAIN** policies, the programmer is able to define a rich number of possible behaviors.

RETAIN and USE_ACTIVE_OBJECT_MAP_ONLY

This combination represents the situation where the **POA** does no automatic object activation (that is, the **POA** searches only the Active Object Map). The server must activate all objects served by the **POA** explicitly, using either the **activate_object** or **activate_object_with_id** operation.

RETAIN and USE_SERVANT_MANAGER

This combination represents a very common situation, where there is an Active Object Map and a **ServantManager**.

Because **RETAIN** is in effect, the application can call **activate_object** or **activate_object_with_id** to establish known servants in the Active Object Map for use in later requests.

If the **POA** doesn't find a servant in the Active Object Map for a given object, it tries to determine the servant by means of invoking incarnate in the **ServantManager** (specifically a ServantActivator) registered with the POA. If no **ServantManager** is available, the **POA** raises the OBJECT_ADAPTER system exception.

RETAIN and USE_DEFAULT_SERVANT

This combination represents the situation where there is a default servant defined for all requests involving unknown objects.

Because **RETAIN** is in effect, the application can call **activate_object** or **activate_object_with_id** to establish known servants in the Active Object Map for use in later requests.

The **POA** first tries to find a servant in the Active Object Map for a given object. If it does not find such a servant, it uses the default servant. If no default servant is available, the **POA** raises the OBJECT_ADAPTER system exception.

NON-RETAIN and USE_SERVANT_MANAGER

This combination represents the situation where one servant is used per method call.

The **POA** doesn't try to find a servant in the Active Object Map because the **ActiveObjectMap** does not exist. In every request, it will call preinvoke on the **ServantManager** (specifically a **ServantLocator**) registered with the **POA**. If no **ServantManager** is available, the **POA** will raise the OBJECT_ADAPTER system exception.

NON-RETAIN and USE_DEFAULT_SERVANT

This combination represents the situation where there is one single servant defined for all CORBA objects.

The **POA** does not try to find a servant in the Active Object Map because the **ActiveObjectMap** doesn't exist. In every request, the **POA** will invoke the appropriate operation on the default servant registered with the **POA**. If no default servant is available, the **POA** will raise the OBJECT_ADAPTER system exception.

11.3.7.7 Implicit Activation Policy

Objects with the **ImplicitActivationPolicy** interface are obtained using the **POA::create_implicit_activation_policy** operation and passed to the **POA::create_POA** operation to specify whether implicit activation of servants is supported in the created POA. The following values can be supplied.

- **IMPLICIT_ACTIVATION** - the POA will support implicit activation of servants. **IMPLICIT_ACTIVATION** also requires the **SYSTEM_ID** and **RETAIN** policies.
- **NO_IMPLICIT_ACTIVATION** - the POA will not support implicit activation of servants.

If no **ImplicitActivationPolicy** is specified at POA creation, the default is **NO_IMPLICIT_ACTIVATION**.

11.3.8 POA Interface

A POA object manages the implementation of a collection of objects. The POA supports a name space for the objects, which are identified by Object Ids.

A POA also provides a name space for POAs. A POA is created as a child of an existing POA, which forms a hierarchy starting with the root POA.

11.3.8.1 Locality Constraints

A **POA** object must not be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a MARSHAL system exception. An attempt to use a **POA** object with the DII may raise the NO_IMPLEMENT exception.

11.3.8.2 create_POA

```
POA create_POA(
    in string               adapter_name,
    in POAManager           a_POAManager,
    in CORBA::PolicyList    policies)
    raises (AdapterAlreadyExists, InvalidPolicy);
```

This operation creates a new POA as a child of the target POA. The specified name identifies the new POA with respect to other POAs with the same parent POA. If the target POA already has a child POA with the specified name, the AdapterAlreadyExists exception is raised.

If the **a_POAManager** parameter is null, a new **POAManager** object is created and associated with the new POA. Otherwise, the specified **POAManager** object is associated with the new POA. The **POAManager** object can be obtained using the attribute name **the_POAManager**.

The specified policy objects are associated with the POA and used to control its behavior. The policy objects are effectively copied before this operation returns, so the application is free to destroy them while the POA is in use. Policies are *not* inherited from the parent POA.

If any of the policy objects specified are not valid for the ORB implementation, if conflicting policy objects are specified, or if any of the specified policy objects require prior administrative action that has not been performed, an InvalidPolicy exception is raised containing the index in the policies parameter value of the first offending policy object.

Note – Creating a POA using a POA manager that is in the active state can lead to race conditions if the POA supports preexisting objects, because the new POA may receive a request before its adapter activator, servant manager, or default servant have been initialized. These problems do not occur if the POA is created by an adapter activator registered with a parent of the new POA, because requests are queued until the adapter activator returns. To avoid these problems when a POA must be explicitly initialized, the application can initialize the POA by invoking **find_POA** with a TRUE activate parameter.

11.3.8.3 find_POA

```
POA find_POA(
    in string       adapter_name,
```

```
    in boolean      activate_it)
    raises (AdapterNonExistent);
```

If the target **POA** is the parent of a child **POA** with the specified name (relative to the target **POA**), that child **POA** is returned. If a child **POA** with the specified name does not exist and the value of the **activate_it** parameter is **TRUE**, the target **POA**'s **AdapterActivator**, if one exists, is invoked, and, if it successfully activates the child **POA**, that child **POA** is returned. Otherwise, the AdapterNonExistent exception is raised.

11.3.8.4 destroy

```
void destroy(
    in boolean          etherealize_objects,
    in boolean          wait_for_completion);
```

This operation destroys the **POA** and all descendant **POAs**. All descendant **POA**s are destroyed (recursively) before the destruction of the containing **POA**. The **POA** so destroyed (that is, the **POA** with its name) may be re-created later in the same process. (This differs from the **POAManager::deactivate** operation that does not allow a re-creation of its associated **POA** in the same process. After a deactivate, re-creation is allowed only if the **POA** is later destroyed.)

When **destroy** is called the **POA** behaves as follows:

- The **POA** calls **destroy** on all of its immediate descendants.
- After all descendant **POA**s have been destroyed and their servants etherealized, the **POA** continues to process requests until there are no requests executing in the **POA**. The apparent destruction of the **POA** occurs only after all executing requests in the **POA** have completed. After destruction has become apparent, the **POA** may be re-created via either an **AdapterActivator** or a call to **create_POA**.
- If the **etherealize_objects** parameter is **TRUE**, the **POA** has the **RETAIN** policy, and a servant manager is registered with the **POA**, the etherealize operation on the servant manager is called for each active object in the Active Object Map. The apparent destruction of the POA occurs before any calls to etherealize are made. Thus, for example, an etherealize method that attempts to invoke operations on the **POA** receives the **OBJECT_NOT_EXIST** exception. Once apparent destruction has occurred, the **POA** behaves as if its **POAManager** is in the holding state until destruction is complete. Thus, for example, an invocation of **create_POA** with the same name blocks until **POA** destruction has finished.

The **wait_for_completion** parameter is handled as follows:

- If **wait_for_completion** is **TRUE** and the current thread is not in an invocation context dispatched from some **POA** belonging to the same ORB as this **POA**, the destroy operation returns only after all active requests have completed and all invocations of **etherealize** have completed.
- If **wait_for_completion** is **TRUE** and the current thread is in an invocation context dispatched from some **POA** belonging to the same ORB as this **POA**, the BAD_INV_ORDER exception is raised and **POA** destruction does not occur.

- If **wait_for_completion** is **FALSE**, the **destroy** operation destroys the POA and its children but waits neither for active requests to complete nor for etherealization to occur. If **destroy** is called multiple times before destruction is complete (because there are active requests), the **etherealize_objects** parameter applies only to the first call of **destroy**. Subsequent calls with conflicting **etherealize_objects** settings use the value of **etherealize_objects** from the first call. The **wait_for_completion** parameter is handled as defined above for each individual call (some callers may choose to block, while others may not).

11.3.8.5 Policy Creation Operations

```
ThreadPolicy create_thread_policy(
        in ThreadPolicyValue value);
LifespanPolicy create_lifespan_policy(
        in LifespanPolicyValue value);
IdUniquenessPolicy create_id_uniqueness_policy(
        in IdUniquenessPolicyValue value);
IdAssignmentPolicy create_id_assignment_policy(
        in IdAssignmentPolicyValue value);
ImplicitActivationPolicy create_implicit_activation_policy(
        in ImplicitActivationPolicyValue value);
ServantRetentionPolicy create_servant_retention_policy(
        in ServantRetentionPolicyValue value);
RequestProcessingPolicy create_request_processing_policy(
        in RequestProcessingPolicyValue value);
```

These operations each return a reference to a policy object with the specified value. The application is responsible for calling the inherited **destroy** operation on the returned reference when it is no longer needed.

11.3.8.6 the_name

```
readonly attribute string the_name;
```

This attribute identifies the POA relative to its parent. This name is assigned when the POA is created. The name of the root POA is system-dependent and should not be relied upon by the application.

11.3.8.7 the_parent

```
readonly attribute POA the_parent;
```

This attribute identifies the parent of the POA. The parent of the root POA is null.

11.3.8.8 the_children

```
readonly attribute POAList the_children;
```

This attribute identifies the current set of all child POAs of the POA. The set of child POAs includes only the POA's immediate children, and not their descendants.

11.3.8.9 *the_POAManager*

```
readonly attribute POAManager the_POAManager;
```

This attribute identifies the POA manager associated with the POA.

11.3.8.10 *the_activator*

```
attribute AdapterActivator the_activator;
```

This attribute identifies the adapter activator associated with the POA. A newly created POA has no adapter activator (the attribute is null). It is system-dependent whether the root POA initially has an adapter activator; the application is free to assign its own adapter activator to the root POA.

11.3.8.11 *get_servant_manager*

```
ServantManager get_servant_manager()
    raises(WrongPolicy);
```

This operation requires the **USE_SERVANT_MANAGER** policy; if not present, the WrongPolicy exception is raised.

If the **ServantRetentionPolicy** of the **POA** is **RETAIN**, then the **ServantManager** argument (**imgr**) shall support the ServantActivator interface (e.g., in C++ **imgr** is narrowable to **ServantActivator**). If the ServantRetentionPolicy of the POA is NON_RETAIN, then the **ServantManager** argument shall support the **ServantLocator** interface. If the argument is **nil**, or does not support the required interface, then the OBJ_ADAPTER exception is raised.

This operation returns the servant manager associated with the POA. If no servant manager has been associated with the POA, it returns a null reference.

11.3.8.12 *set_servant_manager*

```
void set_servant_manager(in ServantManager imgr)
    raises(WrongPolicy);
```

This operation requires the **USE_SERVANT_MANAGER** policy; if not present, the WrongPolicy exception is raised.

This operation sets the default servant manager associated with the POA. This operation may only be invoked once after a POA has been created. Attempting to set the servant manager after one has already been set will result in the BAD_INV_ORDER exception being raised.

11.3.8.13 get_servant

```
Servant get_servant()
    raises(NoServant, WrongPolicy);
```

This operation requires the **USE_DEFAULT_SERVANT** policy; if not present, the WrongPolicy exception is raised.

This operation returns the default servant associated with the POA. If no servant has been associated with the POA, the NoServant exception is raised.

11.3.8.14 set_servant

```
void set_servant(in Servant p_servant)
    raises(WrongPolicy);
```

This operation requires the **USE_DEFAULT_SERVANT** policy; if not present, the WrongPolicy exception is raised.

This operation registers the specified servant with the POA as the default servant. This servant will be used for all requests for which no servant is found in the Active Object Map.

11.3.8.15 activate_object

```
ObjectId activate_object(in Servant p_servant)
    raises (ServantAlreadyActive, WrongPolicy);
```

This operation requires the **SYSTEM_ID** and **RETAIN** policy; if not present, the WrongPolicy exception is raised.

If the POA has the **UNIQUE_ID** policy and the specified servant is already in the Active Object Map, the ServantAlreadyActive exception is raised. Otherwise, the **activate_object** operation generates an Object Id and enters the Object Id and the specified servant in the Active Object Map. The Object Id is returned.

11.3.8.16 activate_object_with_id

```
void activate_object_with_id(
    in          ObjectId oid,
    in Servant  p_servant)
    raises (ObjectAlreadyActive, ServantAlreadyActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the WrongPolicy exception is raised.

If the CORBA object denoted by the Object Id value is already active in this POA (there is a servant bound to it in the Active Object Map), the ObjectAlreadyActive exception is raised. If the POA has the **UNIQUE_ID** policy and the servant is already

in the Active Object Map, the ServantAlreadyActive exception is raised. Otherwise, the **activate_object_with_id** operation enters an association between the specified Object Id and the specified servant in the Active Object Map.

If the **POA** has the **SYSTEM_ID** policy and it detects that the Object Id value was not generated by the system or for this **POA**, the **activate_object_with_id** operation may raise the BAD_PARAM system exception. An ORB is not required to detect all such invalid Object Id values, but a portable application must not invoke **activate_object_with_id** on a **POA** that has the **SYSTEM_ID** policy with an Object Id value that was not previously generated by the system for that **POA**, or, if the **POA** also has the **PERSISTENT** policy, for a previous instantiation of the same **POA**.

11.3.8.17 deactivate_object

```
void deactivate_object(
    in ObjectId oid)
    raises (ObjectNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the WrongPolicy exception is raised.

This operation causes the **ObjectId** specified in the **oid** parameter to be deactivated. An **ObjectId** which has been deactivated continues to process requests until there are no active requests for that **ObjectId**. A deactivated **ObjectId** is removed from the Active Object Map when all requests executing for that **ObjectId** have completed. If a servant manager is associated with the **POA**, **ServantActivator::etherealize** is invoked with the **oid** and the associated servant after the **ObjectId** has been removed from the Active Object Map. Reactivation for the **ObjectId** blocks until etherealization (if necessary) is complete. This includes implicit activation (as described in etherealize) and explicit activation via **POA::activate_object_with_id**. Once an **ObjectId** has been removed from the Active Object Map and etherealized (if necessary) it may then be reactivated through the usual mechanisms.

The operation does not wait for requests or etherealization to complete and always returns immediately after deactivating the **ObjectId**.

Note – If the servant associated with the **oid** is serving multiple Object Ids, **ServantActivator::etherealize** may be invoked multiple times with the same servant when the other objects are deactivated. It is the responsibility of the object implementation to refrain from destroying the servant while it is active with any Id.

11.3.8.18 create_reference

```
Object create_reference (
    in CORBA::RepositoryId intf)
    raises (WrongPolicy);
```

This operation requires the **SYSTEM_ID** policy; if not present, the WrongPolicy exception is raised.

This operation creates an object reference that encapsulates a POA-generated Object Id value and the specified interface repository id. The specified repository id, which may be a null string, will become the **type_id** of the generated object reference. A repository id that does not identify the most derived interface of the object or one of its base interfaces will result in undefined behavior.

This operation does not cause an activation to take place. The resulting reference may be passed to clients, so that subsequent requests on those references will cause the appropriate servant manager to be invoked, if one is available. The generated Object Id value may be obtained by invoking **POA::reference_to_id** with the created reference.

11.3.8.19 create_reference_with_id

```
Object create_reference_with_id (
    in ObjectId                 oid,
    in CORBA::RepositoryId      intf);
```

This operation creates an object reference that encapsulates the specified Object Id and interface repository Id values. The specified repository id, which may be a null string, will become the **type_id** of the generated object reference. A repository id that does not identify the most derived interface of the object or one of its base interfaces will result in undefined behavior.

This operation does not cause an activation to take place. The resulting reference may be passed to clients, so that subsequent requests on those references will cause the object to be activated if necessary, or the default servant used, depending on the applicable policies.

If the **POA** has the **SYSTEM_ID** policy and it detects that the Object Id value was not generated by the system or for this POA, the **create_reference_with_id** operation may raise the BAD_PARAM system exception. An ORB is not required to detect all such invalid Object Id values, but a portable application must not invoke this operation on a POA that has the **SYSTEM_ID** policy with an Object Id value that was not previously generated by the system for that **POA**, or, if the **POA** also has the **PERSISTENT** policy, for a previous instantiation of the same **POA**.

11.3.8.20 servant_to_id

```
ObjectId servant_to_id(
    in Servant                  p_servant)
    raises (ServantNotActive, WrongPolicy);
```

This operation requires the **USE_DEFAULT_SERVANT** policy or a combination of the RETAIN policy and either the **UNIQUE_ID** or **IMPLICIT_ACTIVATION** policies; if not present, the WrongPolicy exception is raised.

This operation has four possible behaviors.

1. If the **POA** has the **UNIQUE_ID** policy and the specified servant is active, the Object Id associated with that servant is returned.

2. If the **POA** has the **IMPLICIT_ACTIVATION** policy and either the POA has the **MULTIPLE_ID** policy or the specified servant is not active, the servant is activated using a POA-generated Object Id and the Interface Id associated with the servant, and that Object Id is returned.

3. If the **POA** has the **USE_DEFAULT_SERVANT** policy, the servant specified is the default servant, and the operation is being invoked in the context of executing a request on the default servant, then the ObjectId associated with the current invocation is returned.

4. Otherwise, the ServantNotActive exception is raised.

11.3.8.21 servant_to_reference

```
Object servant_to_reference (
    in Servant                  p_servant)
    raises (ServantNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy and either the **UNIQUE_ID** or **IMPLICIT_ACTIVATION** policies if invoked outside the context of an operation dispatched by this POA. If this operation is not invoked in the context of executing a request on the specified servant and the policies specified previously are not present the WrongPolicy exception is raised.

This operation has four possible behaviors.

1. If the POA has both the **RETAIN** and the **UNIQUE_ID** policy and the specified servant is active, an object reference encapsulating the information used to activate the servant is returned.

2. If the **POA** has both the **RETAIN** and the **IMPLICIT_ACTIVATION** policy and either the **POA** has the **MULTIPLE_ID** policy or the specified servant is not active, the servant is activated using a POA-generated Object Id and the Interface Id associated with the servant, and a corresponding object reference is returned.

3. If the operation was invoked in the context of executing a request on the specified servant, the reference associated with the current invocation is returned.

4. Otherwise, the ServantNotActive exception is raised.

Note – The allocation of an Object Id value and installation in the Active Object Map caused by implicit activation may actually be deferred until an attempt is made to externalize the reference. The real requirement here is that a reference is produced that will behave appropriately (that is, yield a consistent Object Id value when asked politely).

11.3.8.22 reference_to_servant

```
Servant reference_to_servant (
    in Object                   reference)
    raises (ObjectNotActive, WrongAdapter, WrongPolicy);
```

This operation requires the **RETAIN** policy or the **USE_DEFAULT_SERVANT** policy. If neither policy is present, the WrongPolicy exception is raised.

If the **POA** has the **RETAIN** policy and the specified object is present in the Active Object Map, this operation returns the servant associated with that object in the Active Object Map. Otherwise, if the **POA** has the **USE_DEFAULT_SERVANT** policy and a default servant has been registered with the **POA**, this operation returns the default servant. Otherwise, the ObjectNotActive exception is raised.

If the object reference was not created by this **POA**, the WrongAdapter exception is raised.

11.3.8.23 reference_to_id

```
ObjectId reference_to_id(
    in Object                   reference)
    raises (WrongAdapter, WrongPolicy);
```

The WrongPolicy exception is declared to allow future extensions.

This operation returns the Object Id value encapsulated by the specified **reference**. This operation is valid only if the reference was created by the POA on which the operation is being performed. If the reference was not created by that POA, a WrongAdapter exception is raised. The object denoted by the reference does not have to be active for this operation to succeed.

11.3.8.24 id_to_servant

```
Servant id_to_servant(
    in ObjectId                 oid)
    raises (ObjectNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy or the **USE_DEFAULT_SERVANT** policy. If neither policy is present, the WrongPolicy exception is raised.

If the POA has the **RETAIN** policy and the specified ObjectId is in the Active Object Map, this operation returns the servant associated with that object in the Active Object Map. Otherwise, if the POA has the **USE_DEFAULT_SERVANT** policy and a default servant has been registered with the POA, this operation returns the default servant. Otherwise the ObjectNotActive exception is raised.

11.3.8.25 id_to_reference

```
Object id_to_reference(
    in ObjectId                 oid)
    raises (ObjectNotActive, WrongPolicy);
```

This operation requires the **RETAIN** policy; if not present, the WrongPolicy exception is raised.

If an object with the specified Object Id value is currently active, a reference encapsulating the information used to activate the object is returned. If the Object Id value is not active in the POA, an ObjectNotActive exception is raised.

11.3.9 Current operations

The **PortableServer::Current** interface, derived from **CORBA::Current**, provides method implementations with access to the identity of the object on which the method was invoked. The Current interface is provided to support servants that implement multiple objects, but can be used within the context of POA-dispatched method invocations on any servant. To provide location transparency, ORBs are required to support use of **Current** in the context of both locally and remotely invoked operations.

An instance of Current can be obtained by the application by issuing the **CORBA::ORB::resolve_initial_references("POACurrent")** operation. Thereafter, it can be used within the context of a method dispatched by the **POA** to obtain the **POA** and **ObjectId** that identify the object on which that operation was invoked.

11.3.9.1 get_POA

```
POA get_POA()
    raises (NoContext);
```

This operation returns a reference to the POA implementing the object in whose context it is called. If called outside the context of a POA-dispatched operation, a NoContext exception is raised.

11.3.9.2 get_object_id

```
ObjectId get_object_id()
    raises (NoContext);
```

This operation returns the ObjectId identifying the object in whose context it is called. If called outside the context of a POA-dispatched operation, a NoContext exception is raised.

11.4 IDL for PortableServer module

```
#pragma prefix "omg.org"
module PortableServer {
#   pragma version PortableServer 2.3
    interface POA;                  // forward declaration
    typedef sequence<POA> POAList;

    native Servant;

    typedef sequence<octet> ObjectId;
```

```
exception ForwardRequest {
    Object forward_reference;
};

// Policy interfaces

const CORBA::PolicyType THREAD_POLICY_ID = 16;
const CORBA::PolicyType LIFESPAN_POLICY_ID = 17;
const CORBA::PolicyType ID_UNIQUENESS_POLICY_ID = 18;
const CORBA::PolicyType ID_ASSIGNMENT_POLICY_ID = 19;
const CORBA::PolicyType IMPLICIT_ACTIVATION_POLICY_ID = 20;
const CORBA::PolicyType SERVANT_RETENTION_POLICY_ID = 21;
const CORBA::PolicyType REQUEST_PROCESSING_POLICY_ID = 22;

enum ThreadPolicyValue {
    ORB_CTRL_MODEL,
    SINGLE_THREAD_MODEL
};

interface ThreadPolicy : CORBA::Policy {
    readonly attribute ThreadPolicyValue value;
};

enum LifespanPolicyValue {
    TRANSIENT,
    PERSISTENT
};

interface LifespanPolicy : CORBA::Policy {
    readonly attribute LifespanPolicyValue value;
};

enum IdUniquenessPolicyValue {
    UNIQUE_ID,
    MULTIPLE_ID
};

interface IdUniquenessPolicy : CORBA::Policy {
    readonly attribute IdUniquenessPolicyValue value;
};

enum IdAssignmentPolicyValue {
    USER_ID,
    SYSTEM_ID
};

interface IdAssignmentPolicy : CORBA::Policy {
    readonly attribute IdAssignmentPolicyValue value;
};

enum ImplicitActivationPolicyValue {
```

```
        IMPLICIT_ACTIVATION,
        NO_IMPLICIT_ACTIVATION
};

interface ImplicitActivationPolicy : CORBA::Policy {
        readonly attribute ImplicitActivationPolicyValue value;
};

enum ServantRetentionPolicyValue {
        RETAIN,
        NON_RETAIN
};

interface ServantRetentionPolicy : CORBA::Policy {
        readonly attribute ServantRetentionPolicyValue value;
};

enum RequestProcessingPolicyValue {
        USE_ACTIVE_OBJECT_MAP_ONLY,
        USE_DEFAULT_SERVANT,
        USE_SERVANT_MANAGER
};

interface RequestProcessingPolicy : CORBA::Policy {
        readonly attribute RequestProcessingPolicyValue value;
};

// POAManager interface

interface POAManager {
        exception AdapterInactive{};

        enum State {HOLDING, ACTIVE, DISCARDING, INACTIVE};

        void activate()
                raises(AdapterInactive);
        void hold_requests(
                in boolean wait_for_completion)
                raises(AdapterInactive);
        void discard_requests(
                in boolean wait_for_completion)
                raises(AdapterInactive);
        void deactivate(
                in boolean etherealize_objects,
                in boolean wait_for_completion)
                raises(AdapterInactive);
        State get_state();
};

// AdapterActivator interface
```

```
    interface AdapterActivator {
#       pragma version AdapterActivator 2.3
        boolean unknown_adapter(
            in POA parent,
            in string name);
        };

    // ServantManager interface

    interface ServantManager{ };

    interface ServantActivator : ServantManager {
#       pragma version ServantActivator 2.3
        Servant incarnate (
            in ObjectId             oid,
            in POA                  adapter)
        raises (ForwardRequest);

        void etherealize (
            in ObjectId             oid,
            in POA                  adapter,
            in Servant              serv,
            in boolean              cleanup_in_progress,
            in boolean              remaining_activations);
        };

    interface ServantLocator : ServantManager {
#       pragma version ServantLocator 2.3
        native Cookie;
        Servant preinvoke(
            in ObjectId             oid,
            in POA                  adapter,
            in CORBA::Identifier    operation,
            out Cookie              the_cookie)
        raises (ForwardRequest);

        void postinvoke(
            in ObjectId             oid,
            in POA                  adapter,
            in CORBA::Identifier    operation,
            in Cookie               the_cookie,
            in Servant              the_servant
        );
    };

    // POA interface

    interface POA {
#       pragma version POA 2.3
        exception AdapterAlreadyExists {};
        exception AdapterNonExistent {};
```

```
exception InvalidPolicy {unsigned short index;};
exception NoServant {};
exception ObjectAlreadyActive {};
exception ObjectNotActive {};
exception ServantAlreadyActive {};
exception ServantNotActive {};
exception WrongAdapter {};
exception WrongPolicy {};

// POA creation and destruction

POA create_POA(
    in string adapter_name,
    in POAManager a_POAManager,
    in CORBA::PolicyList policies)
raises (AdapterAlreadyExists, InvalidPolicy);

POA find_POA(
    in string adapter_name,
    in boolean activate_it)
raises (AdapterNonExistent);

void destroy(
    in boolean etherealize_objects,
    in boolean wait_for_completion);

// Factories for Policy objects

ThreadPolicy create_thread_policy(
    in ThreadPolicyValue value);
LifespanPolicy create_lifespan_policy(
    in LifespanPolicyValue value);
IdUniquenessPolicy create_id_uniqueness_policy(
    in IdUniquenessPolicyValue value);
IdAssignmentPolicy create_id_assignment_policy(
    in IdAssignmentPolicyValue value);
ImplicitActivationPolicy create_implicit_activation_policy(
    in ImplicitActivationPolicyValue value);
ServantRetentionPolicy create_servant_retention_policy(
    in ServantRetentionPolicyValue value);
RequestProcessingPolicy create_request_processing_policy(
    in RequestProcessingPolicyValue value);

// POA attributes

readonly attribute string the_name;
readonly attribute POA the_parent;
readonly attribute POAList the_children;
readonly attribute POAManager the_POAManager;
attribute AdapterActivator the_activator;
```

```
// Servant Manager registration:

ServantManager get_servant_manager()
raises (WrongPolicy);

void set_servant_manager(
    in ServantManager imgr)
raises (WrongPolicy);

// operations for the USE_DEFAULT_SERVANT policy

Servant get_servant()
raises (NoServant, WrongPolicy);

void set_servant(in Servant p_servant)
raises (WrongPolicy);

// object activation and deactivation

ObjectId activate_object(
    in Servant p_servant)
raises (ServantAlreadyActive, WrongPolicy);

void activate_object_with_id(
    in ObjectId id,
    in Servant p_servant)
raises (ServantAlreadyActive, ObjectAlreadyActive, WrongPolicy);

void deactivate_object(
    in ObjectId oid)
raises (ObjectNotActive, WrongPolicy);

// reference creation operations

Object create_reference (
    in CORBA::RepositoryId intf)
raises (WrongPolicy);

Object create_reference_with_id (
    in ObjectId oid,
    in CORBA::RepositoryId intf)
raises (WrongPolicy);

// Identity mapping operations:

ObjectId servant_to_id(
    in Servant p_servant)
raises (ServantNotActive, WrongPolicy);

Object servant_to_reference(
    in Servant p_servant)
```

```
            raises (ServantNotActive, WrongPolicy);

            Servant reference_to_servant(
                in Object reference)
            raises(ObjectNotActive, Wrongpolicy);

            ObjectId reference_to_id(
                in Object reference)
            raises (WrongAdapter, WrongPolicy);

            Servant id_to_servant(
                in ObjectId oid)
            raises (ObjectNotActive, WrongPolicy);

            Object id_to_reference(in ObjectId oid)
                raises (ObjectNotActive, WrongPolicy);
        };

        // Current interface

        interface Current : CORBA::Current {
#           pragma version Current 2.3
            exception NoContext { };

            POA get_POA()
            raises (NoContext);

            ObjectId get_object_id()
                raises (NoContext);
        };
};
```

11.5 UML Description of PortableServer

The following diagrams were generated by an automated tool and then annotated with the cardinalities of the associations. They are intended to be an aid in comprehension to those who enjoy such representations. They are not normative.

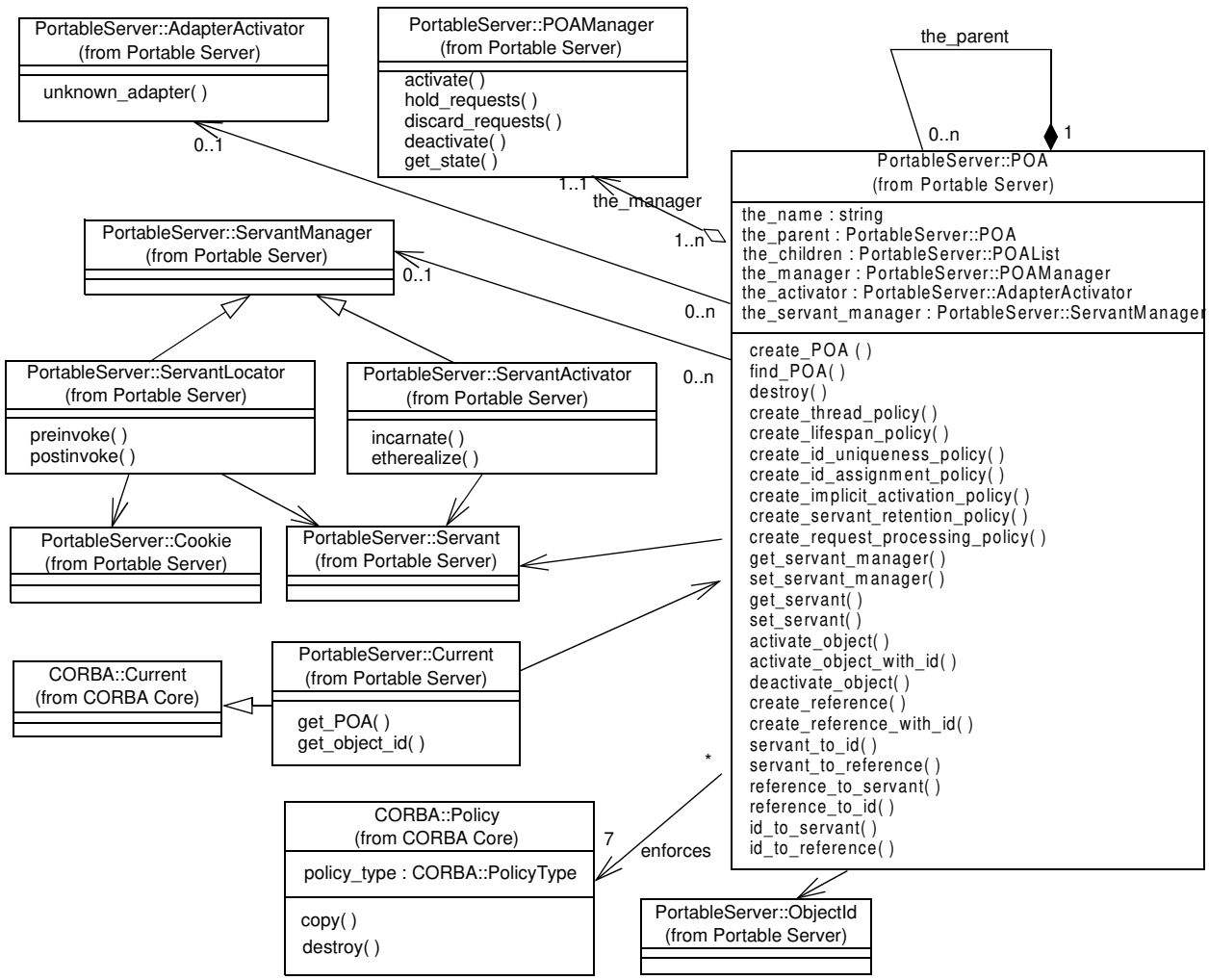

Figure 11-4 UML for main part of PortableServer

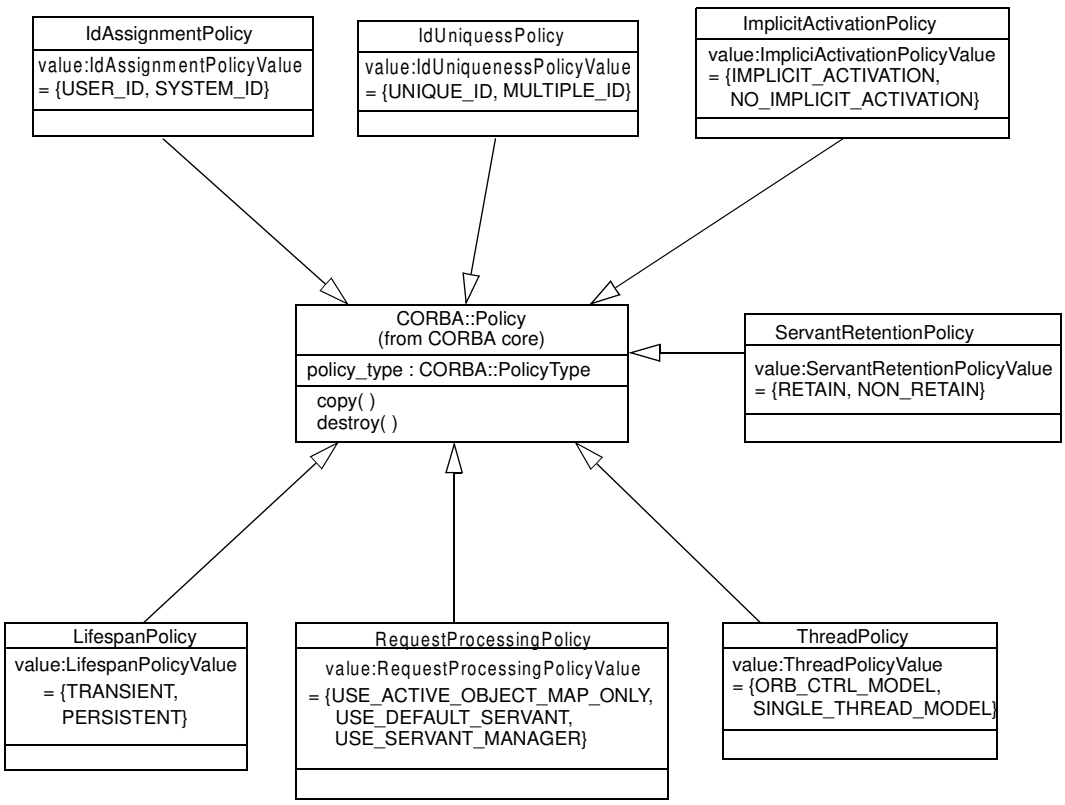

Figure 11-5 UML for PortableServer policies

11.6 Usage Scenarios

This section illustrates how different capabilities of the POA may be used in applications.

Note – In some of the following C++ examples, PortableServer names are not explicitly scoped. It is assumed that all the examples have the C++ statement
`using namespace PortableServer;`

11.6.1 Getting the root POA

All server applications must obtain a reference to the root POA, either to use it directly to manage objects, or to create new POA objects. The following example demonstrates how the application server can obtain a reference to the root POA.

```
// C++
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
CORBA::Object_ptr pfobj =
orb->resolve_initial_references("RootPOA");
```

```
PortableServer::POA_ptr rootPOA;
rootPOA = PortableServer::POA::narrow(pfobj);
```

11.6.2 Creating a POA

For a variety of reasons, a server application might want to create a new POA. The POA is created as a child of an existing POA. In this example, it is created as a child of the root POA.

```
// C++
CORBA::PolicyList policies(2);
policies.length(2);
policies[0] = rootPOA->create_thread_policy(
PortableServer::ThreadPolicy::ORB_CTRL_MODEL);
policies[1] = rootPOA->create_lifespan_policy(
PortableServer::LifespanPolicy::TRANSIENT);
PortableServer::POA_ptr poa =
rootPOA->create_POA("my_little_poa",
PortableServer::POAManager::_nil(), policies);
```

11.6.3 Explicit Activation with POA-assigned Object Ids

By specifying the **SYSTEM_ID** policy on a POA, objects may be explicitly activated through the POA without providing a user-specified identity value. Using this approach, objects are activated by performing the **activate_object** operation on the POA with the object in question. For this operation, the POA allocates, assigns, and returns a unique identity value for the object.

Generally this capability is most useful for transient objects, where the Object Id needs to be valid only as long as the servant is active in the server. The Object Ids can remain completely hidden and no servant manager need be provided. When this is the case, the identity and lifetime of the servant and the abstract object are essentially equivalent. When POA-assigned Object Ids are used with persistent objects or objects that are activated on demand, the application must be able to associate the generated Object Id value with its corresponding object state.

This example illustrates a simple implementation of transient objects using POA-assigned Object Ids. It presumes a POA that has the **SYSTEM_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

Assume this interface:

```
// IDL
    interface Foo {
        long doit();
    };
```

This might result in the generation of the following skeleton:

```
class POA_Foo : public ServantBase
{
   public:
      ...
   virtual CORBA::Long doit() = 0;
}
```

Derive your implementation:

```
class MyFooServant : public POA_Foo
{
   public:
      MyFooServant(POA_ptr poa, Long value)
      : my_poa(POA::_duplicate(poa)), my_value(value) {}
      ~MyFooServant() {CORBA::release(my_poa);}
      virtual POA_ptr _default_POA()
         {return POA::_duplicate(my_poa);}
      virtual Long doit() {return my_value;}
   protected:
      POA_ptr my_poa;
      Long my_value;
};
```

Now, somewhere in the program during initialization, probably in **main()**:

```
   MyFooServant* afoo = new MyFooServant(poa,27);
   PortableServer::ObjectId_var oid =
      poa->activate_object(afoo);
   Foo_var foo = afoo->_this();
   poa->the_POAManager()->activate();
   orb->run();
```

This object is activated with a generated Object Id.

11.6.4 Explicit Activation with User-assigned Object Ids

An object may be explicitly activated by a server using a user-assigned identity. This may be done for several reasons. For example, a programmer may know that certain objects are commonly used, or act as initial points of contact through which clients access other objects (for example, factories). The server could be implemented to create and explicitly activate these objects during initialization, avoiding the need for a servant manager.

If an implementation has a reasonably small number of servants, the server may be designed to keep them all active continuously (as long as the server is executing). If this is the case, the implementation need not provide a servant manager. When the server initializes, it could create all available servants, loading their state and identities from some persistent store. The POA supports an explicit activation operation, **activate_object_with_id**, that associates a servant with an Object Id. This operation would be used to activate all of the existing objects managed by the server during server initialization. Assuming the POA has the **USE_SERVANT_MANAGER** policy

and no servant manager is associated with a POA, any request received by the POA for an Object Id value not present in the Active Object Map will result in an **OBJECT_ADAPTER** exception.

In simple cases of well-known, long-lived objects, it may be sufficient to activate them with well-known Object Id values during server initialization, before activating the POA. This approach ensures that the objects are always available when the POA is active, and doesn't require writing a servant manager. It has severe practical limitations for a large number of objects, though.

This example illustrates the explicit activation of an object using a user-chosen Object Id. This example presumes a POA that has the **USER_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

The code is like the previous example, but replace the last portion of the example shown above with the following code:

```
// C++
MyFooServant* afoo = new MyFooServant(poa, 27);
PortableServer::ObjectId_var oid =
   PortableServer::string_to_ObjectId("myLittleFoo");
poa->activate_object_with_id(oid.in(), afoo);
Foo_var foo = afoo->_this();
```

11.6.5 Creating References before Activation

It is sometimes useful to create references for objects before activating them. This example extends the previous example to illustrate this option:

```
// C++
PortableServer::ObjectId_var oid =
PortableServer::string_to_ObjectId("myLittleFoo");
CORBA::Object_var obj = poa->create_reference_with_id(
      oid.in(), "IDL:Foo:1.0");
Foo_var foo = Foo::_narrow(obj);

// ...later...
MyFooServant* afoo = new MyFooServant(poa, 27);
poa->activate_object_with_id(oid.in(), afoo);
```

11.6.6 Servant Manager Definition and Creation

Servant managers are object implementations, and are required to satisfy all of the requirements of object implementations necessary for their intended function. Because servant managers are local objects, and their use is limited to a single narrow role, some simplifications in their implementation are possible. Note that these simplifications are suggestions, not normative requirements. They are intended as examples of ways to reduce the programming effort required to define servant managers.

A servant manager implementation must provide the following things:

- implementation code for either
  - **incarnate( )** and **etherealize( )**, or
  - **preinvoke( )** and **postinvoke( )**
- implementation code for the servant operations, as for all servants

The first two are obvious; their content is dictated by the requirements of the implementation that the servant manager is managing. For the third point, the default servant manager on the root POA already supplies this implementation code. User-written servant managers will have to provide this themselves.

Since servant managers are objects, they themselves must be activated. It is expected that most servant managers can be activated on the root POA with its default set of policies (see “POA Creation” on page 11-6). It is for this reason that the root POA has the **IMPLICIT_ACTIVATION** policy so that a servant manager can easily be activated. Users may choose to activate a servant manager on other POAs.

The following is an example servant manager to activate objects on demand. This example presumes a POA that has the **USER_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

Since RETAIN is in effect, the type of servant manager used is a ServantActivator. The ORB supplies a servant activator skeleton class in a library:

```
// C++
namespace POA_PortableServer
{
   class ServantActivator : public virtual ServantManager
   {
      public:
         virtual ~ServantActivator();
         virtual Servant incarnate(
            const ObjectId& POA_ptr poa) = 0;
         virtual void etherealize(
            const ObjectId&, POA_ptr poa,
            Servant, Boolean remaining_activations) = 0;
   };
};
```

A ServantActivator servant manager might then look like:

```
// C++
class MyFooServantActivator : public
   POA_PortableServer::ServantActivator
{
   public:
      // ...
      Servant incarnate(
         const ObjectId& oid, POA_ptr poa)
      {
```

```
            String_var s = PortbleServer::ObjectId_to_string
                           (oid);
            if (strcmp(s, "myLittleFoo") == 0) {
               return new MyFooServant(poa, 27);
            else {
               throw CORBA::OBJECT_NOT_EXIST();
            }
         }

         void etherealize(
                     const ObjectId& oid,
                     POA_ptr poa,
                     Servant servant,
                     Boolean remaining_activations)
         {
            if (remaining_activations == 0)
               delete servant;
         }
         // ...
};
```

11.6.7 Object Activation on Demand

The precondition for this scenario is the existence of a client with a reference for an object with which no servant is associated at the time the client makes a request on the reference. It is the responsibility of the ORB, in collaboration with the POA and the server application to find or create an appropriate servant and perform the requested operation on it. Such an object is said to be *incarnated* (or *incarnation*) when it has an active servant. Note that the client had to obtain the reference in question previously from some source. From the client's perspective, the abstract object exists as long as it holds a reference, until it receives an OBJECT_NOT_EXIST system exception in a reply from an attempted request on the object. Incarnation state does not imply existence or non-existence of the abstract object.

Note – This specification does not address the issues of communication or server process activation, as they are immaterial to the POA interface and operation. It is assumed that the ORB activates the server if necessary, and can deliver the request to the appropriate POA.

To support object activation on demand, the server application must register a servant manager with the appropriate POA. Upon receiving the request, if the POA consults the Active Object Map and discovers that there is no active servant associated with the target Object Id, the POA invokes the **incarnate** operation on the servant manager.

Note – An implication that this model has for GIOP is that the object key in the request message must encapsulate the Object Id value. In addition, it may encapsulate other values as necessitated by the ORB implementation. For example, the server must be able to determine to which POA the request should be directed. It could assign a different communication endpoint to each POA so that the POA identity is implicit in the request, or it could use a single endpoint for the entire server and encapsulate POA identities in object key values. Note that this is not a concrete requirement; the object key may not actually contain any of those values. Whatever the concrete information is, the ORB and POA must be able to use it to find the servant manager, invoke activate if necessary (which requires the actual Object Id value), and/or find the active servant in some map.

The **incarnate** invocation passes the Object Id value to the servant manager. At this point, the servant manager may take any action necessary to produce a servant that it considers to be a valid incarnation of the object in question. The operation returns the servant to the POA, which invokes the operation on it. The **incarnate** operation may alternatively raise an OBJECT_NOT_EXIST system exception that will be returned to the invoking client. In this way, the user-supplied implementation is responsible for determining object existence and non-existence.

After activation, the POA maintains the association of the servant and the Object Id in the Active Object Map. (This example presumes the **RETAIN** and **USE_SERVANT_MANAGER** policies.)

As an obvious example of transparent activation, the Object Id value could contain a key for a record in a database that contains the object's state. The servant manager would retrieve the state from the database, construct a servant of the appropriate implementation class (assuming an object-oriented programming language), initialize it with the state from the database, and return it to the POA.

The example servant manager in the last section ("Servant Manager Definition and Creation" on page 11-52) could be used for this scenario. Recall that the POA would have the **USER_ID**, **USE_SERVANT_MANAGER**, and **RETAIN** policies.

Given such a ServantActivator, all that remains is initialization code such as the following.

```
PortableServer::ObjectId_var oid =
   PortableServer::string_to_ObjectId("myLittleFoo");
CORBA::Object_var obj = poa->create_reference_with_id(
   oid, "IDL:foo:1.0");
MyFooServantActivator* fooIM = new MyFooServantActivator;
ServantActivator_var IMref = fooIM->_this();
poa->set_servant_manager(IMref);
poa->the_POAmanager()->activate();
orb->run();
```

11.6.8 Persistent Objects with POA-assigned Ids

It is possible to access the Object Id value assigned to an object by the POA, with the **POA::reference_to_id** operation. If the reference is for an object managed by the POA that is the operation's target, the operation will return the Object Id value, whether it was assigned by the POA or the user. By doing this, an implementation may provide a servant manager that associates the POA-allocated Object Id values with persistently stored state. It may also pass the POA-allocated Object Id values to POA operations such as **activate_object_with_id** and **create_reference_with_id**.

A POA with the **PERSISTENT** policy may be destroyed and later reinstantiated in the same or a different process. A POA with both the **SYSTEM_ID** and **PERSISTENT** policies generates Object Id values that are unique across all instantiations of the same POA.

11.6.9 Multiple Object Ids Mapping to a Single Servant

Each POA is created with a policy that indicates whether or not servants are allowed to support multiple object identities simultaneously. If a POA allows multiple identities per servant, the POA's treatment of the servants is affected in the following ways:

- Servants of the type may be explicitly activated multiple times with different identity values without raising an exception.
- A servant cannot be mapped onto or converted to an individual object reference using that POA, since the identity is potentially ambiguous.

11.6.10 One Servant for all Objects

By using the **USE_DEFAULT_SERVANT** policy, the developer can create a POA that will use a single servant to implement all of its objects. This approach is useful when there is very little data associated with each object, so little that the data can be encoded in the Object Id.

The following example illustrates this approach by using a single servant to incarnate all CORBA objects that export a given interface in the context of a server. This example presumes a POA that has the **USER_ID**, **NON_RETAIN,** and **USE_DEFAULT_SERVANT** policies.

Two interfaces are defined in IDL. The **FileDescriptor** interface is supported by objects that will encapsulate access to operations in a file associated with a file system. Global operations in a file system, such as the ones necessary to create **FileDescriptor** objects, are supported by objects that export the **FileSystem** interface.

```
// IDL
interface FileDescriptor {
    typedef sequence<octet> DataBuffer;

    long write (in DataBuffer buffer);
    DataBuffer read (
```

```
            in long num_bytes);
        void destroy ();
    };

    interface FileSystem {
        ...
        FileDescriptor open (
            in string file_name,
            in long flags);
        ...
    };
```

Implementation of these two IDL interfaces may inherit from static skeleton classes generated by an IDL to C++ compiler as follows:

```
// C++
class FileDescriptorImpl : public POA_FileDescriptor
{
   public:
      FileDescriptorImpl(POA_ptr poa);
      ~FileDescriptorImpl();
      POA_ptr _default_POA();
      CORBA::Long write(
         const FileDescriptor::DataBuffer& buffer);
      FileDescriptor::DataBuffer* read(
         CORBA::Long num_bytes);
      void destroy();
   private:
      POA_ptr my_poa;
};

class FileSystemImpl : public POA_FileSystem
{
   public:
      FileSystemImpl(POA_ptr poa);
      ~FileSystemImpl();
      POA_ptr _default_POA();
      FileDescriptor_ptr open(
         const char* file_name, CORBA::Long flags);
   private:
      POA_ptr my_poa;
      FileDescriptorImpl* fd_servant;
};
```

A single servant may be used to serve all requests issued to all **FileDescriptor** objects created by a **FileSystem** object. The following fragment of code illustrates the steps to perform when a **FileSystem** servant is created.

```
// C++
FileSystemImpl::FileSystemImpl(POA_ptr poa)
    : my_poa(POA::_duplicate(poa))
```

```
{
   fd_servant = new FileDescriptorImpl(poa);
   poa->set_servant(fd_servant);
};
```

The following fragment of code illustrates how **FileDescriptor** objects are created as a result of invoking an operation (**open**) exported by a **FileSystem** object. First, a local file descriptor is created using the appropriate operating system call. Then a CORBA object reference is created and returned to the client. The value of the local file descriptor will be used to distinguish the new **FileDescriptor** object from other **FileDescriptor** objects. Note that FileDescriptor objects in the example are transient, since they use the value of their file descriptors for their ObjectIds, and of course the file descriptors are only valid for the life of a process.

```
// C++
FileDescriptor_ptr
FileSystemImpl::open(
   const char* file_name, CORBA::Long flags)
{
   int fd = ::open(file_name, flags);
   ostrstream ostr;
   ostr << fd;
   PortableServer::ObjectId_var oid =
   PortableServer::string_to_ObjectId(ostr.str());
   Object_var obj = my_poa->create_reference_with_id(
      oid.in(),"IDL:FileDescriptor:1.0");
   return FileDescriptor::_narrow(obj);
};
```

Any request issued to a **FileDescriptor** object is handled by the same servant. In the context of a method invocation, the servant determines which particular object is being incarnated by invoking an operation that returns a reference to the target object and, after that, invoking **POA::reference_to_id**. In C++, the operation used to obtain a reference to the target object is **_this()**. Typically, the **ObjectId** value associated with the reference will be used to retrieve the state of the target object. However, in this example, such a step is not required since the only thing that is needed is the value for the local file descriptor and that value coincides with the **ObjectId** value associated with the reference.

Implementation of the **read** operation is rather simple. The servant determines which object it is incarnating, obtains the local file descriptor matching its identity, performs the appropriate operating system call, and returns the result in a **DataBuffer** sequence.

```
// C++
FileDescriptor::DataBuffer*
FileDescriptorImpl::read(CORBA::Long num_bytes)
{
   FileDescriptor_var me = _this();
   PortableServer::ObjectId_var oid =
      my_poa->reference_to_id(me.in());
   CORBA::String_var s =
```

```
        PortableServer::ObjectId_to_string(oid.in());
    istrstream is(s);
    int fd;
    is >> fd;
    CORBA::Octet* buffer = DataBuffer::alloc_buf(num_bytes);
    int len = ::read(fd, buffer, num_bytes);
    DataBuffer* result = new DataBuffer(len, len, buffer, 1);
    return result;
};
```

Using a single servant per interface is useful in at least two situations.

- In one case, it may be appropriate for encapsulating access to legacy APIs that are not object-oriented (system calls in the Unix environment, as we have shown in the example).
- In another case, this technique is useful in handling scalability issues related to the number of CORBA objects that can be associated with a server. In the example above, there may be a million **FileDescriptor** objects in the same server and this would only require one entry in the ORB. Although there are operating system limitations in this respect (a Unix server is not able to open so many local file descriptors) the important point to take into account is that usage of CORBA doesn't introduce scalability problems but provides mechanisms to handle them.

11.6.11 Single Servant, Many Objects and Types, Using DSI

The ability to associate a single DSI servant with many CORBA objects is rather powerful in some scenarios. It can be the basis for development of gateways to legacy systems or software that mediates with external hardware, for example.

Usage of the DSI is illustrated in the following example. This example presumes a POA that supports the **USER_ID**, **USE_DEFAULT_SERVANT** and **RETAIN** policies.

A single servant will be used to incarnate a huge number of CORBA objects, each of them representing a separate entry in a Database. There may be several types of entries in the Database, representing different entity types in the Database model. Each type of entry in the Database is associated with a separate interface which comprises operations supported by the Database on entries of that type. All these interfaces inherit from the **DatabaseEntry** interface. Finally, an object supporting the **DatabaseAgent** interface supports basic operations in the database such as creating a new entry, destroying an existing entry, etc.

```
// IDL
    interface DatabaseEntry {
        readonly attribute string name;
    };


    interface Employee : DatabaseEntry {
        attribute long id;
```

```
        attribute long salary;
    };
...

    interface DatabaseAgent {
        DatabaseEntry create_entry (
            in string key,
            in CORBA::Identifier entry_type,
            in NVPairSequence initial_attribute_values
        );

    void destroy_entry (
        in string key);
        ...
};
```

Implementation of the **DatabaseEntry** interface may inherit from the standard dynamic skeleton class as follows:

```
// C++
class DatabaseEntryImpl :
   public PortableServer::DynamicImplementation
{
   public:
      DatabaseEntryImpl (DatabaseAccessPoint db);
      virtual void invoke (ServerRequest_ptr request);
      ~DatabaseEntryImpl ();

      virtual POA_ptr _default_POA()
      {
         return poa;
      }
};
```

On the other hand, implementation of the **DatabaseAgent** interface may inherit from a static skeleton class generated by an IDL to C++ compiler as follows:

```
// C++
class DatabaseAgentImpl :
   public DatabaseAgentImplBase
{
   protected:
      DatabaseAccessPoint mydb;
      DatabaseEntryImpl * common_servant;
   public:
      DatabaseAgentImpl ();
      virtual DatabaseEntry_ptr create_entry (
         const char * key,
         const char * entry_type,
         const NVPairSequence& initial_attribute_values
      );
```

```
        virtual void destroy_entry (const char * key);
        ~DatabaseAgentImpl ();
};
```

A single servant may be used to serve all requests issued to all **DatabaseEntry** objects created by a **DatabaseAgent** object. The following fragment of code illustrates the steps to perform when a **DatabaseAgent** servant is created. First, access to the database is initialized. As a result, some kind of descriptor (a DatabaseAccessPoint local object) used to operate on the database is obtained. Finally, a servant will be created and associated with the POA.

```
// C++
void DatabaseAgentImpl::DatabaseAgentImpl ()
{
   mydb = ...;
   common_servant = new DatabaseEntryImpl(mydb);
   poa->set_servant(common_servant);
};
```

The code used to create **DatabaseEntry** objects representing entries in the database is similar to the one used for creating **FileDescriptor** objects in the example of the previous section. In this case, a new entry is created in the database and the key associated with that entry will be used to represent the identity for the corresponding **DatabaseEntry** object. All requests issued to a **DatabaseEntry** object are handled by the same servant because references to this type of object are associated with a common POA created with the **USE_DEFAULT_SERVANT** policy.

```
// C++
DatabaseEntry_ptr DatabaseAgentImpl::create_entry (
   const char * key,
   const char * entry_type,
   const NVPairSequence& initial_attribute_values)

   // creates a new entry in the database:
   mydb->new_entry (key, ...);

   // creates a reference to the CORBA object used to
   // encapsulate access to the new entry in the database.
   // There is an interface for each entry type:
   CORBA::Object_ptr obj = poa->create_reference_with_id(
      string_to_ObjectId (key),
      identifierToRepositoryId (entry_type),
   );

   DatabaseEntry_ptr entry = DatabaseEntry::_narrow (obj);
      CORBA::release (obj);
   return entry;
};
```

Any request issued to a **DatabaseEntry** object is handled by the same servant. In the context of a method invocation, the servant determines which particular object it is incarnating, obtains the database key matching its identity, invokes the appropriate operation in the database and returns the result as an output parameter in the **ServerRequest** object.

Sometimes, a program may need to determine the type of an entry in the database in order to invoke operations on the entry. If that is the case, the servant may obtain the type of an entry based on the interface supported by the **DatabaseEntry** object encapsulating access to that entry. This interface may be obtained by means of invoking the **get_interface** operation exported by the reference to the **DatabaseEntry** object.

```
// C++
void DatabaseEntryImpl::invoke (ServerRequest_ptr request)
{
   CORBA::Object_ptr current_obj = _this ();

   // The servant determines the key associated with
   // the database entry represented by current_obj:
   PortableServer::ObjectId oid =
      poa->reference_to_id (current_obj);
   char * key = ObjectId_to_string (oid);

   // The servant handles the incoming CORBA request. This
   // typically involves the following steps:
   // 1. mapping the CORBA request into a database request
   //    using the key obtained previously
   // 2. constructing output parameters to the CORBA request
   //    from the response to the database request
      ...
};
```

Note that in this example, we may have a billion **DatabaseEntry** objects in a server requiring only a single entry in map tables supported by the POA (that is, the ORB at the server). No permanent storage is required for references to DatabaseEntry objects at the server. Actually, references to DatabaseEntry objects will only occupy space:

- at clients, as long as those references are used; or
- at the server, only while a request is being served.

Scalability problems can be handled using this technique. There are many scenarios where this scalability causes no penalty in terms of performance (basically, when there is no need to restore the state of an object, each time a request to it is being served).