CHAPTER 8 Locking Summary

8.1 Overview

This chapter gives a summary of how server objects are protected by locks, and how multiple threads use locks to synchronize access to shared objects.

8.2 Locking Terminology

Let's define a few terms before plunging into a summary of the locking strategies for the MTX server. For a more complete description of the terms, see the POSIX 1003.4a standard.

Critical section: A segment of shared code or shared object that must be protected.

Lock: An interface for protecting a critical section. Locks can be implemented with a mutex, a semaphore, test and set, or a combination of the these.

Mutual exclusion: A policy that prevents multiple concurrent activities from accessing the same critical section at the same time.

Mutex: In the Mach kernel and in kernels adhering to the POSIX 1003.4a standard, *MUT*ual *EX*clusion is implemented via the Mutex construct. A mutex is just one way of

implementing a lock. A mutex can be built using Dijkstra's P and V operations, spin locks, sequenced locks, unsequenced locks, etc.

Lock granularity: Defines the length of time, and/or the quantity of critical section protected by a lock. The following figure shows the granularity scale applied to the locking of shared objects. Granularity scales from locking a small set (very fine grained) to locking the entire application (very coarse grained).

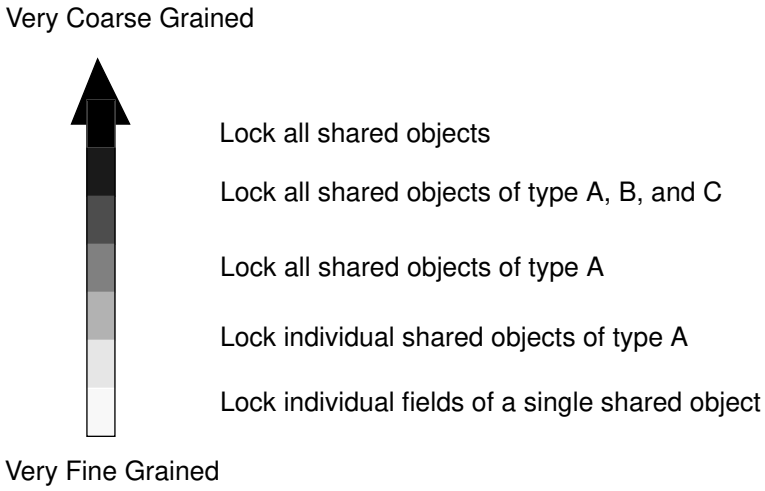


FIGURE 18 Lock Granularity for Shared Objects

Fine Grained locking: If the size of the critical section is small, then holding the associated lock can be brief. Also, if the lock protects parts of an object or a single object out of a set of shared objects, then contention for the lock can be low. If either of these cases exist, we say that the lock is fine grained.

Coarse Grained locking: If the size of the critical section is large, then holding the associated lock can take a proportionally longer time. Also, if the lock protects many shared objects out of a set of objects, then the lock protects a coarse partition of the total set of objects. In this case, contention for the lock can be high since the lock protects many objects. If either of these cases exist, we say that the lock is coarse grained.

Explicit locking: In the MTX server, this defines a lock implemented with a mutex. While an explicit lock is held, the kernel is managing the protection via the mutex construct. An example of an explicit lock is the mutex in a RDB Lock (see CHAPTER 9).

Implicit locking: In the MTX server, this defines a lock implemented with a flag, or a bit in a mask. Since implicit locks are simply application dependent flags, they are protected in turn by an explicit lock. Implicit locks are built by MTX so that an explicit lock doesn't have to be held for a long time. An example of an implicit lock is the Lockbits field in the resource lock record. Note that the RDBLock protects access to the Lockbits field, but the Lockbits protect access to the actual object (see CHAPTER 9).

Lock Conflict: When a thread is holding a lock (whether explicit or implicit), and another thread wants that lock, there is a lock conflict. The thread that holds the lock will block other threads wanting access to the lock.

Conflict Avoidance: It can be expensive for a thread to encounter a lock conflict. The thread must wait until the contended lock is made available. When the thread holding the lock no longer needs it, the thread needs to know if there are threads waiting on the lock so that it can wake them up. Lock conflict can cause threads to wait, can cause increased thread context switching, requires the kernel to maintain which threads are waiting on which lock, and requires threads that give up locks to determine if there are waiting threads to be signalled. Since lock conflicts are expensive, it would be a good idea to avoid them if practical. This can be done by making the lock more fine grained. Threads are less likely to collide over small critical sections. An example of conflict avoidance is the CM_X_RENDER conflict mask bit. If two protocol requests have this bit set, a coarse grained implementation of render locking would be to allow only one request to render at a time. A finer grained solution would be to allow another check to determine if the two requests are using overlapping regions. By avoiding the conflict at the CM_X_RENDER level and performing a more detailed check at the region level, we optimize thread operations (see CHAPTER 10).

8.3 Server Objects

Objects are defined to be any instantiations of data in the server. The MTX server must allow for the efficient and concurrent access of objects by threads. In particular, the locking design must address the following:

1. New Client management - new clients access authentication and authorization objects before instantiation in the MTX server. Protection is provided by the ConnectionMutex.
2. RDB management - all sharable objects located via the Resource DataBase (RDB) are called resources. Resources include windows, pixmaps, GCs, fonts, colormaps, cursors and passive grabs. A hard problem is how to lock windows, since they are organized into a hierarchically based structure (i.e. loosely defined as a tree). In addition, the protection of different resources may require both fine and coarse grained locks. This issue is addressed by the Resource Database (RDB) Monitor (CHAPTER 9) and the Pending Operation Queue (POQ) Monitor (CHAPTER 10).
3. Font management - fonts are looked up in the RDB, but access to the actual object is protected by the Font Object (FO) Monitor. The FO Monitor also protects the OS Font Info and the DIX Font Info objects. See CHAPTER 13 for a discussion of this issue.
4. Other Object management - objects that are not looked up in the RDB include selections, properties, atoms, and screens. These objects are referenced outside the scope of the RDB. This issue is addressed by the Pending Operation Queue (POQ) Monitor (CHAPTER 10), DeviceEvent (DE) Monitor (CHAPTER 11), and other monitors (CHAPTER 13).
5. Rendering - access to the correct region of the display. See the POQ Monitor (CHAPTER 10) and a discussion of the DDX issues (CHAPTER 14).

6. Device database management and Event propagation - Devices may be configured in the server at any time. Although only device events are propagated in the window tree, the MTX server must insure that other server events, including extension events, are protected. These objects are protected by the DeviceEvent (DE) Monitor (CHAPTER 11).
7. Message delivery to clients - replies, errors, and events must be delivered on time and in correct order to specific X clients. Protection is provided by the Message Object (MO) Monitor (CHAPTER 12).
8. DDX specific locks
9. Thread synchronization - the MST must pend until the server begins cleanup, while the DIT must be allowed to change device configuration at any time.
10. Thread specific - each thread has specific information that is implicitly protected because it is private to the thread. For the CIT, this includes parts of the client record. Although the ClientTable is an array of pointers to the ClientRecs, other CITs only read the public part of the ClientRec. The private part of the ClientRec is only referenced by the owning CIT.

FIGURE 19 illustrates some of the major server objects, both shared and non-shared, along with the monitor that protects them. The actual operation of the monitors differ based on the type of object to be locked. For example, the RDB Monitor must manage read, write and exclusive access to a wide variety of shared objects, while the DE Monitor only has to protect the Device Event database. In some cases, multiple locks are needed to access an object. For example, the RDB Monitor is used to lookup a colormap, but the POQ Monitor is used to do the actual locking ofCHAPTER9 the colormap resource. In the diagram, multiple locks are indicated by the object being protected by multiple monitors.

A more complete description of which object is protected by which lock is presented in the next section, and in FIGURE 20.

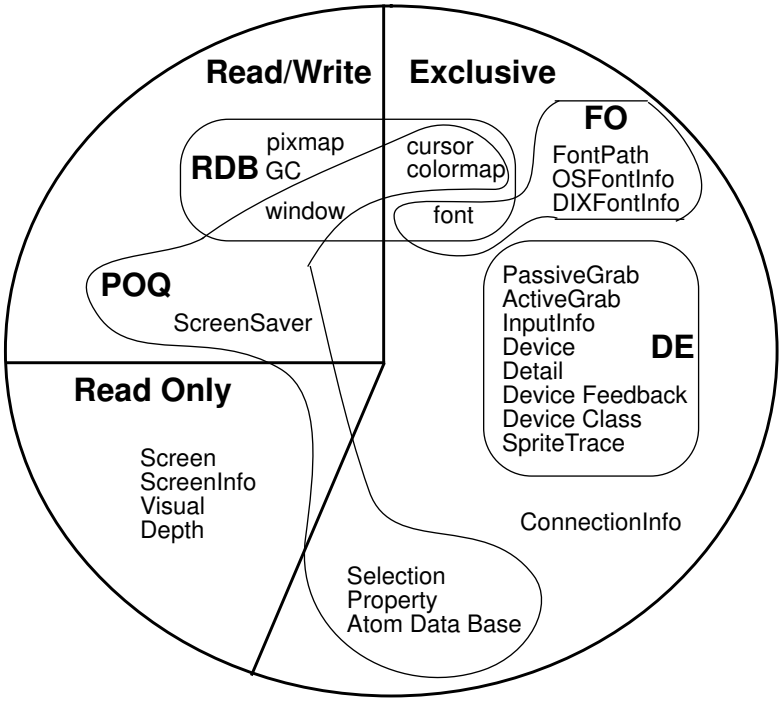


FIGURE 19 Server Object Summary

8.4 Objects protected by locks

The following table summarizes how a server object is protected by a lock. For MTX objects, locks are implemented with a mutex (explicit) or a flag (implicit).

The table notes which locks are used by monitors to enforce protection policies. Reading horizontally across a row of the table, one can determine all the locks needed to access a given object (usually one). Reading vertically down a column of the table, one can determine how many different objects a lock protects. The purpose of the locks in the table is to avoid conflict. In some cases though, another level of conflict avoidance is used (as indicated by the *). For example, when CM_R/W_Hierarchy and CM_R/W_Geometry bits conflict, a lower level check is used to potentially avoid subtree conflicts in the window tree (see CHAPTER 10).

The locks are listed in the order they should be acquired. This lock precedence is needed to enforce deadlock avoidance. For example, the RDBLock[i] is always acquired before the POQLock, and the POQLock always acquired before the DeviceEventLock.

| Monitor | | RDB | | POQ | | | | | | | | | | | | FO | DE | MO | | | | | | | | | | | |

8.5 Thread Lock Dependencies

The following sections describe how MTX threads lock server objects. The timelines in the figures indicate how long a mutex is held. There are two ways to lock objects: explicitly and implicitly.

An object is locked explicitly if a mutex is actually held during the entire time the object is accessed. An example of this kind is when the POQMutex is acquired to manipulate the POQ list. In the following figures, explicit locking is indicated by vertically striped boxes.

An object is locked implicitly if the mutex is acquired only to set bits, flags, or reference counts and is then released. By setting a bit under the protection of the mutex, an object may be implicitly locked without actually holding the mutex for a long time. An example of this kind of locking is when the RDBMutex is used to set the LockBits in a pixmap. The RDBMutex is only held long enough to set the LockBits, and thus can allow other threads to use the mutex. In the following figures, implicit locking is indicated by grayed boxes. This segregation of locking allows for the implementation of multiple levels of granularity.

In the following diagrams, conditional execution of code is indicated by an enclosing dotted box. In these cases, locks may also be conditionally acquired.

8.5.1 MST

The MST uses the three synchronization mutexes to control execution flow. The ServerMutex is held at all times during initialization and cleanup. It is only released when the MST enters a wait state between initialization and cleanup. The MST waits on the ServerCondition variable which is associated with the ServerMutex. The DITMutex is held while performing DIT operations during initialization and cleanup. The MST will wait for a DIT to terminate after initiating a destroy. Likewise, the MST will wait for all clients to terminate after initiating a KillAllClients during cleanup.

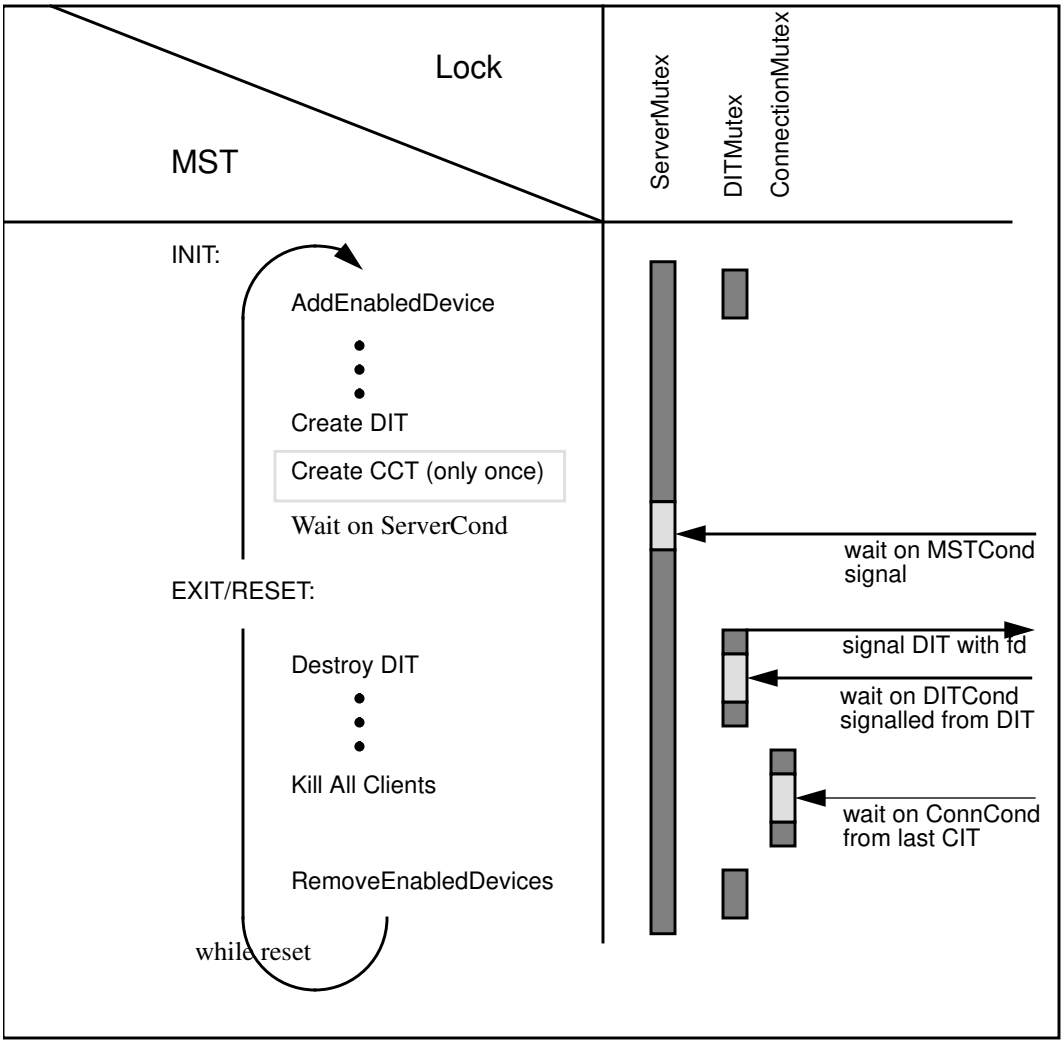


FIGURE 21 MST Locking Summary

8.5.2 CCT

The CCT is designed to hand off new client connections as quickly as possible to the newly created CIT. The CCT uses the ConnectionMutex to protect access to the connection specific select masks. The ServerMutex is obtained before processing a new connection to assure that we do not process a new connection while the server is resetting itself or termininating.

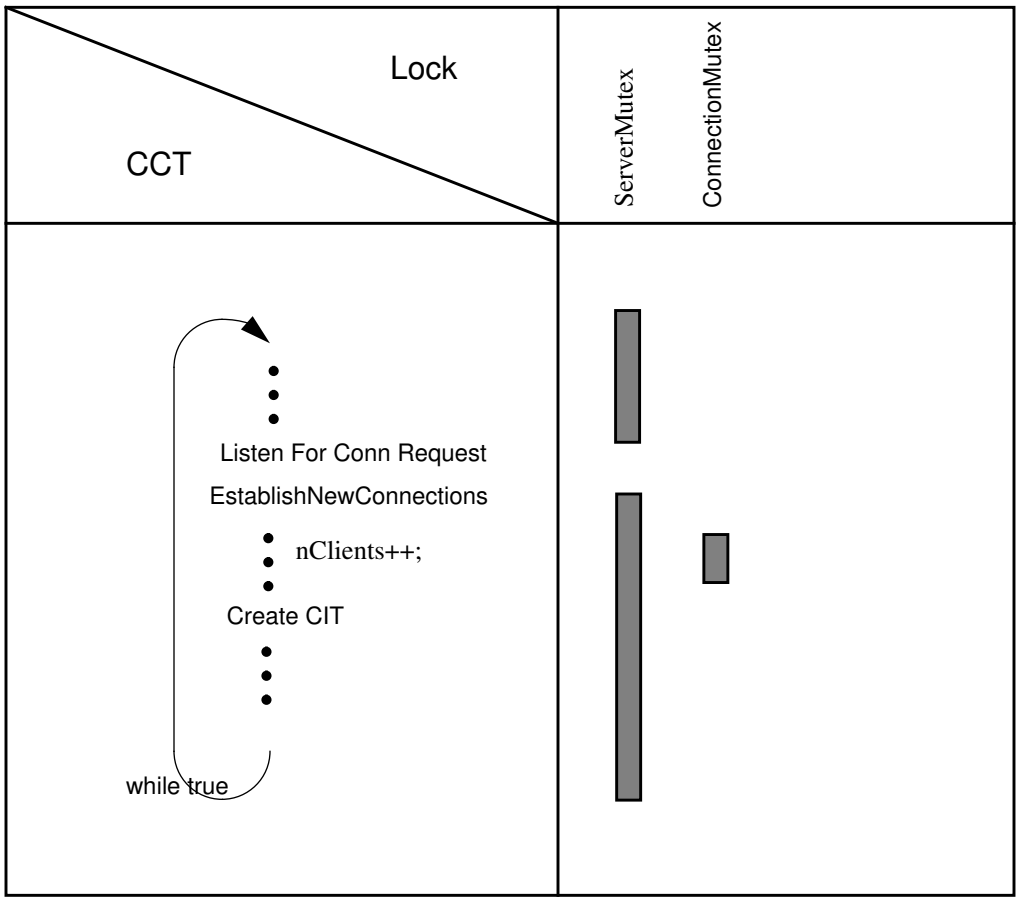


FIGURE 22 CCT Locking Summary

8.5.3 CIT

The CIT performs the largest number of lock/unlocks in the MTX server. Since there is one CIT for each client connected to the server, it is likely that collisions will occur between CITs.

During initialization, the CIT locks the ConnectionMutex while performing connection validation.

During normal operation, the CIT performs the following activities:

- lock resources
- insert element on the POQ
- operate on the protocol request
- remove element from the POQ
- unlock resources

The CIT conditionally looks to see if there are any messages to flush to a client(s). These flush operations are protected by the MO Monitor mutexes. When the CIT dispatches the protocol request, the top level of the protocol request function examines the request to determine which client's RDB resources to lock. The client index ***i*** is used to acquire the RDBMutex[i], so that the CIT can get a handle on the actual object and manipulate resource's lockbits if fine grained locking is needed. The CIT holds the explicit RDBMutex[i] lock for a short time, but may set the resources's lockbits if implicit locking is needed. After locking the RDB resources specified in the request, the CIT acquires the explicit POQMutex just long enough to register the conflict mask (CM) bits, and resolve POQ conflicts. By registering the CM bits, an implicit coarse grained lock is held for the life of the protocol request. The implicit locks are not released until the POQMutex is reacquired at the conclusion of the protocol request. During the protocol request, other contended resources may be accessed, such as fonts, the Device Event Database, or DDX objects. Protocol requests may access these objects in any order, but that order is not always deterministic. So, these locks are acquired as needed and held for as short a time as possible. If the protocol request would generate messages, these messages are queued up in the local buffer of the CIT. They become flushable at the conclusion of the protocol request.

If the CIT must exit, any messages are transferred to the global buffer under the MO Monitor. During the CloseDownCIT function, the server is ungrabbed if necessary. The POQMutex is used to protect access to the global GrabServer objects. Before the client's resources are freed, and all active grabs released, an element is inserted on the POQ using the POQ Monitor. After the CIT is closed down, a final flush to client is performed via the MO Monitor. Finally, the MO Monitor insures that the CIT's local buffer is cleaned up correctly, and the polled messages returned to the global message pool.

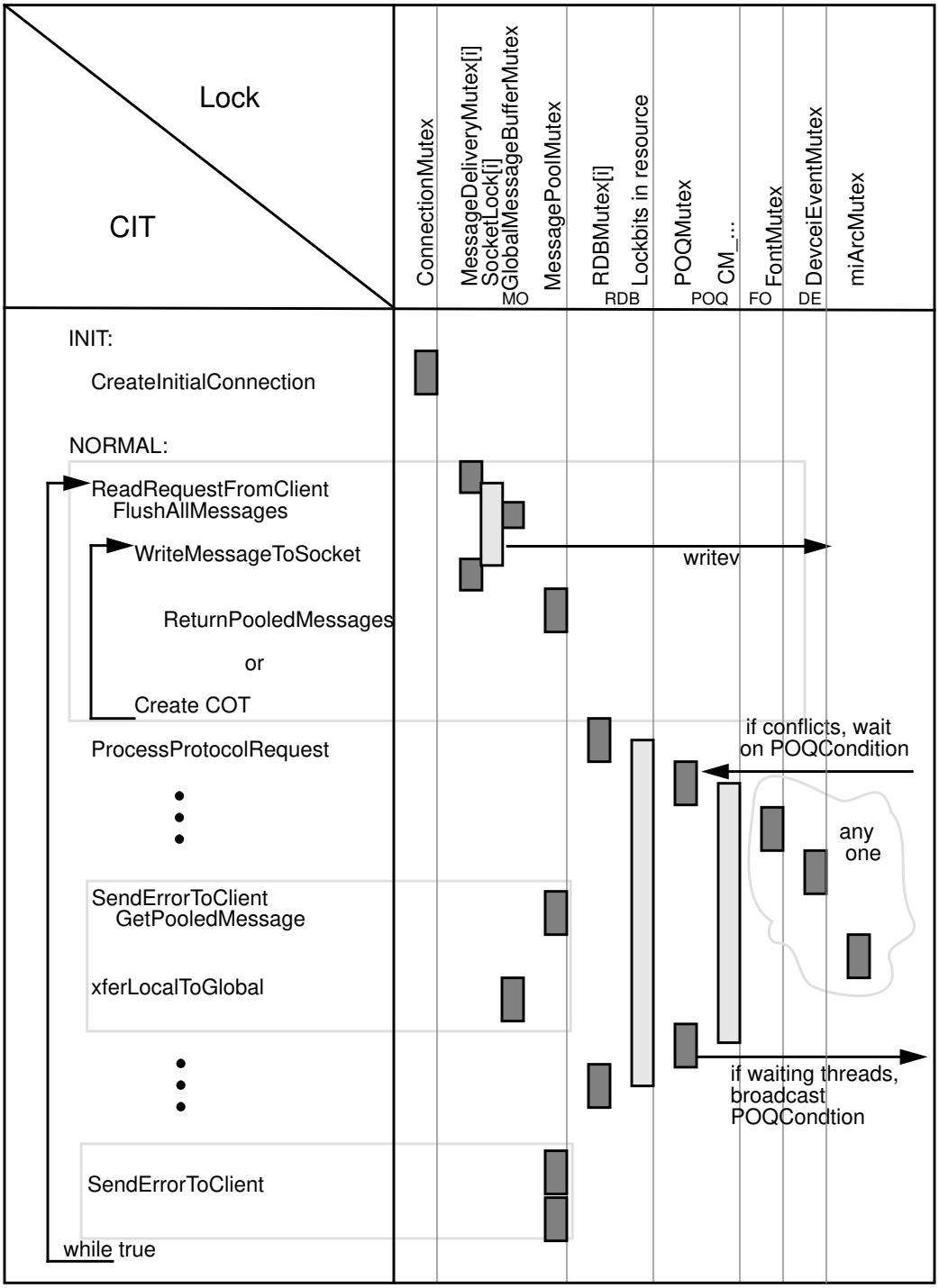


FIGURE 23 CIT Locking Summary (INIT & NORMAL phases)

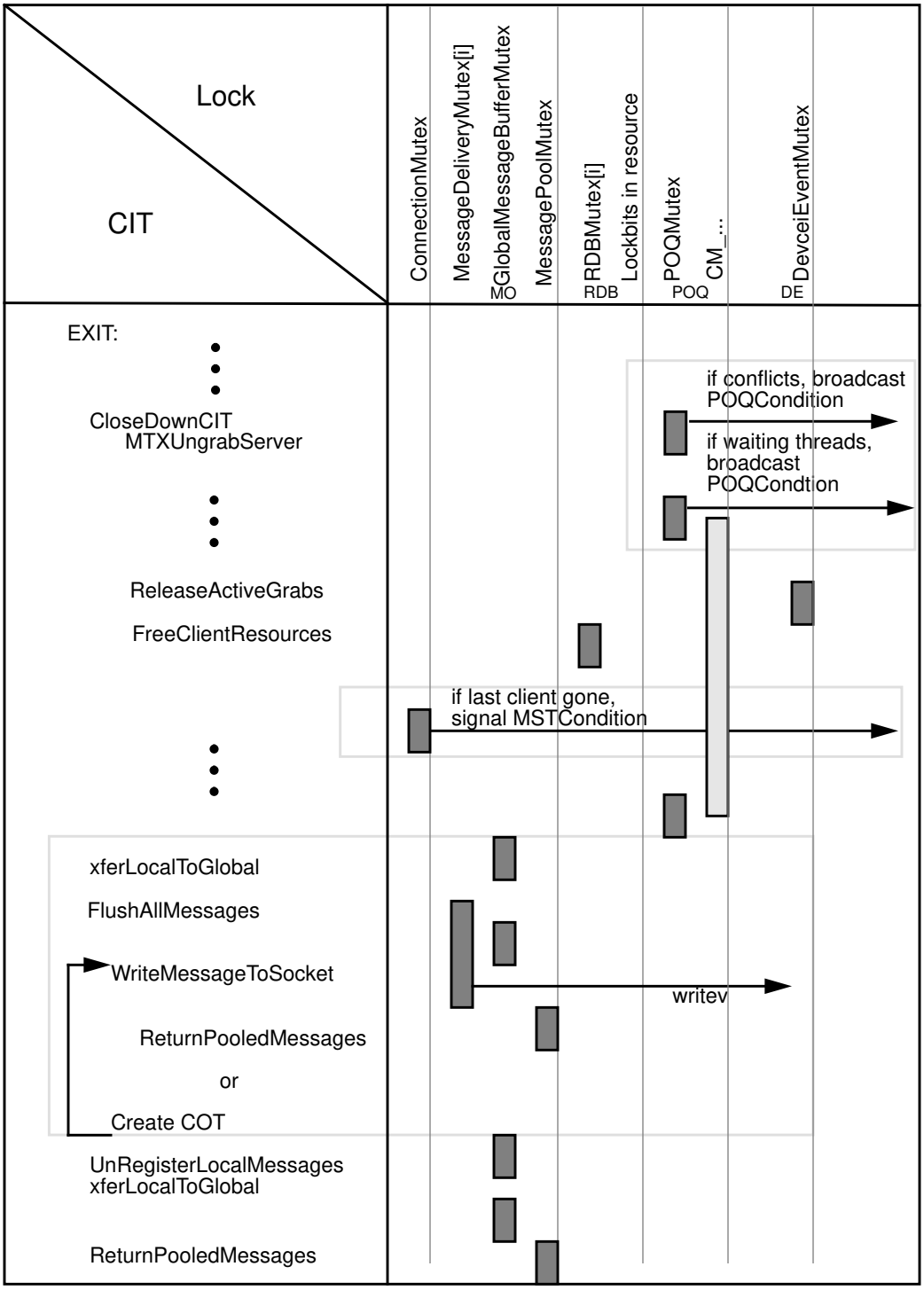


FIGURE 24 CIT Locking Summary (EXIT phase)

8.5.4 COT

The COT is created by the MO Monitor on behalf of a DIT or a CIT to write messages from the global message buffer to the client's socket. The MO Monitor protects access to the global buffer and the socket(s). When the COT is finished with the socket writes, it exits.

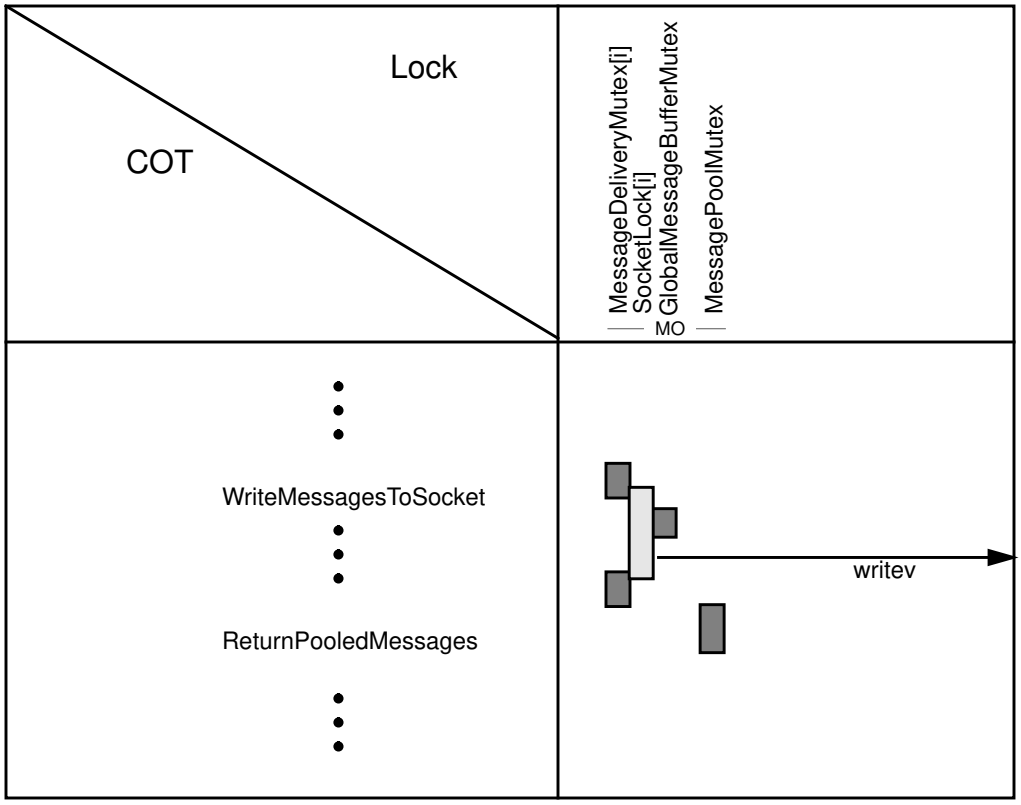


FIGURE 25 COT Locking Summary

8.5.5 DIT

The DIT initialization code sets the screen saver through the POQ Monitor.

During normal operation, the DIT first uses the DITMutex to modify a global select mask before entering the select() function call. If the device wakeup file descriptor indicates that the screen saver state has changed, then the DIT places an entry on the POQ. If the file descriptor is instead associated with a device, then the DIT places itself on the POQ and starts processing of device input. While processing device input, the DIT will access the device event database using the DE Monitor. If any messages are generated (most likely they will be), the DIT will transfer the messages to the global message

buffer using the MO Monitor, and then create a COT to manage the actual transfer to the socket(s).

During DIT shutdown, the local message buffers are freed and the messages returned to the global message pool. The DIT then signals the MST that the ServerCondition is true. This tells the MST to continue server cleanup.

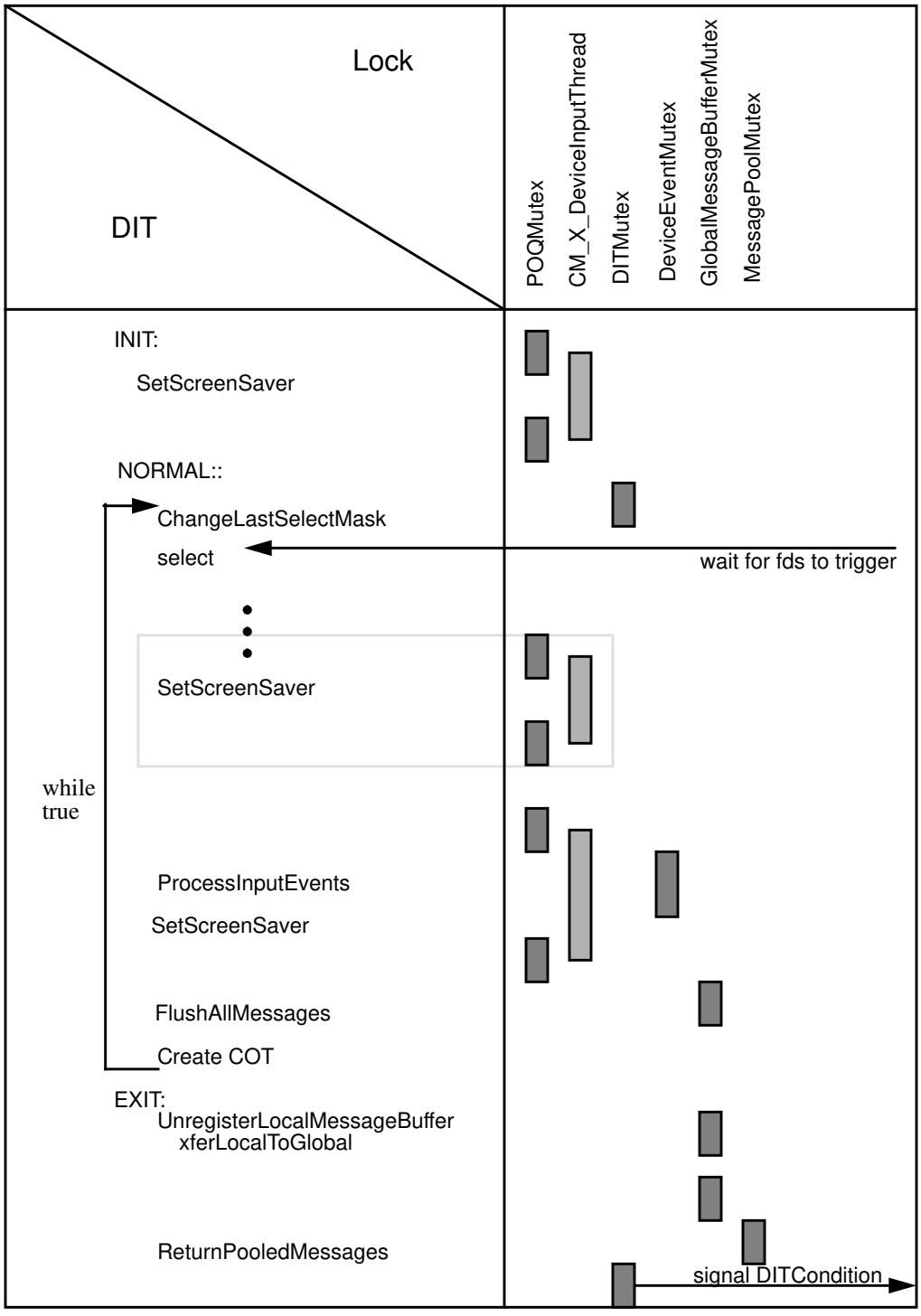


FIGURE 26 DIT Locking Summary

CHAPTER 9 Resource Data Base Monitor

9.1 Overview

The Resource Data Base (RDB) consists of a collection of resources; such as windows, pixmaps, and fonts, created on behalf of a client or the MTX server, and organized into a database that provides quick access to resource data given a resource handle. Clients issue requests that create resources in the server and then maintain handles to them for future access.

The RDB Monitor provides an interface to the RDB and arbitrates access to the internal database structures and the individual resources. It provides all required locking to protect concurrent access to the internal structures and various mechanisms for locking individual resources. FIGURE 27 shows a view of the RDB monitor and its relationship

to the main component, the RDB, along with the various categories of routines available in the external interface.

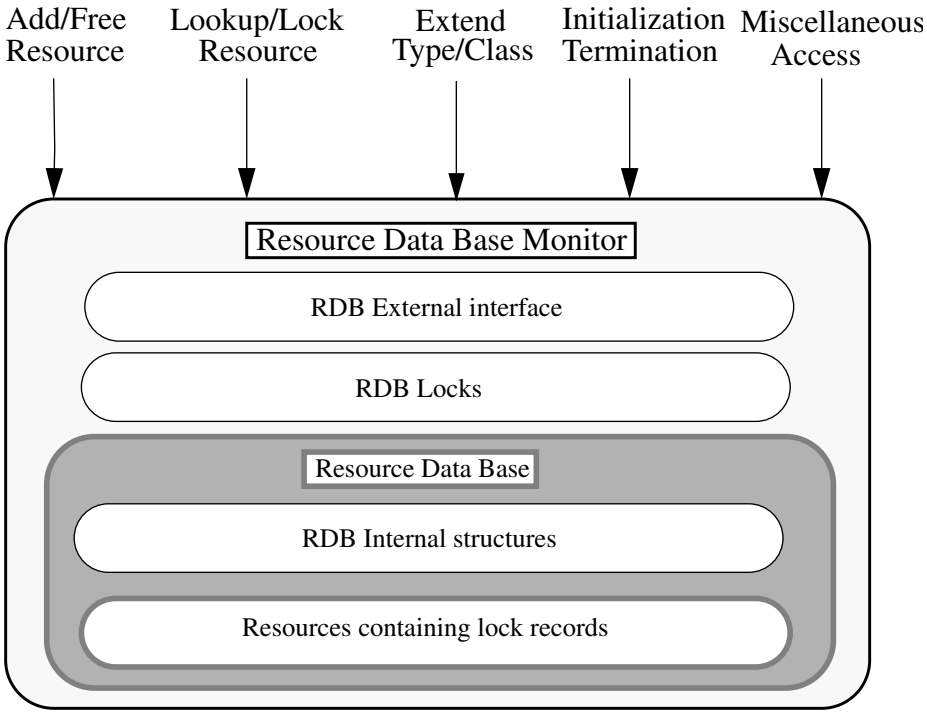


FIGURE 27 Resource Data Base Monitor.

9.2 Data Objects

9.2.1 Resources

X resources are created in the server on behalf of a client. The RDB monitor requires that each resource have an associated type, id, and resource lock. Optionally, the resource may also have an associated class. Every resource in the server will have a unique combination of resource type and resource id.

Resource types and classes: There is a predefined set of core resource types in the MTX server that include such things as windows, pixmaps, cursors, and fonts (see FIGURE 29 for a complete list of all core resource types). This set of resource types can be extended beyond the core types. An X-extension, for instance, may want to use the RDB to manage new resources, and thus, would have to create new resource types for them. Any resource type can claim to be a member of any of the defined resource classes. Associating a resource class with a resource merely provides the opportunity to group this resource with a more general set of resources in the RDB. When attempting to locate and lock a resource via the RDB monitor, a resource type or class must be known. A resource class is normally only used when the exact resource type cannot be determined in the current context.

Resource id: An id is selected for a resource at the time the resource is created. The resource id is unique among all resources of the same type.

Resource lock record: The resource lock record is new for MTX and must be contained within all resources that require the ability to be locked individually. Most resources will be locked individually and require this lock record. A mention of why certain resources may not require individual locking is included in Section 9.3.1. For details on the resource lock and the locking mechanisms, see Section 9.5.1.

Resources are created outside the RDB Monitor and then added to it. The resource class/type and the resource id are not a part of the resource data, rather just associated with it in the RDB. The resource lock, on the other hand, is included as part of the resource data, and is the only portion of the data that is of interest to the RDB monitor. The RDB must know how to locate the lock record in the resource data, but requires no other knowledge about the structure of the individual resources. The byte offset of the lock record within the resource data is associated with each resource type, thus providing the RDB the ability to locate the lock record of a resource given its type. This byte offset must be provided when creating new resource types.

FIGURE 28 shows the relationship between the resource data, resource lock record, and the additional pieces of resource information in the RDB.

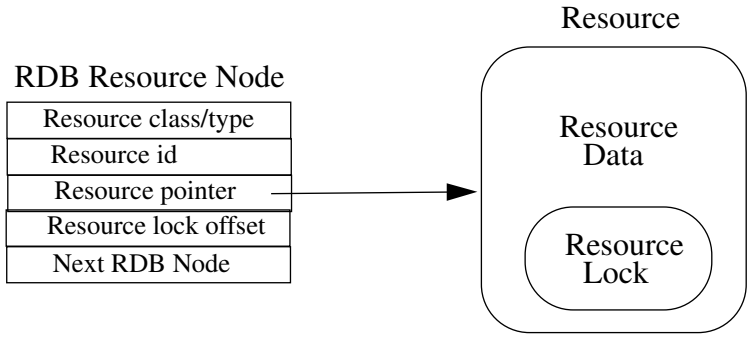


FIGURE 28 Resource and associated RDB node.

In most cases, there is one RDB resource node for each resource in the RDB. However, there are cases where the RDB will contain multiple RDB resource nodes all containing a pointer to the same resource. This is done as a space optimization. A font is an example of a resource with this characteristic. Several clients may each have their own unique resource id for the same server resource. These types of resources will be referred to as **common resources**. For locking purposes, the RDB must be aware of resource types that allow their individual resources to be common (indicated during the creation of new resource types).

Common resources usually have a reference count associated with them indicating how many RDB resource nodes are associated with that resource. The use of a reference count indicates a passive interest in the resource. All resources have an indication of the number of threads with active interest, which are contained in the lock record and dis-

cussed in Section 9.5.1.2. Active interest in a resource implies that the resource is locked and is currently being accessed.

An attempt to remove a common resource from the RDB will result in an RDB resource node being removed and the reference count being decremented. When the reference count becomes zero, the resource data will be deallocated. NOTE: The reference count indicates that pointers to this resource are maintained in structures other than the RDB resource nodes, perhaps even outside the resource database. For instance, a GC resource contains a pointer to a font in the resource data. Accessing resources via one of these pointers should only occur if the resource is properly locked or is known to be a read-only resource.

9.2.2 Resource Data Base Internal Structures

The internal structures of RDB are private to the monitor. They consist of hash tables, RDB resource node lists, resource type tables, resource type/class masks, lookup caches, and RDB lock records. The details and locking issues are discussed in Section 9.5.1.

9.3 Concurrency Issues

The RDB monitor executes within the context of the MST, a CIT, or a DIT. The MST will initialize the monitor and create server resources. A CIT or DIT will often perform add, free, lookup/lock, and unlock operations on resources in the RDB.

The most common function of the RDB monitor is to locate and lock individual resources in the database for use by a protocol request. The locking mechanism used for a resource will depend on its type and its intended access mode.Access to resources in the RDB will not be granted without locking the resource. That is, there is no lookup mechanism without locking.

9.3.1 Resource Lock Types

Each resource type in the RDB will have an associated lock type. All resources of a particular type will be locked with the mechanism defined for the associated lock type. The following lock types are available:

- **exclusive_lock**: allows only one concurrent read or write access to an individual resource. All threads attempting to obtain an exclusive lock when the resource is currently locked will pend. A thread that is pending on a resource will be awakened when releasing the exclusive lock held for that resource.
- **readers_writers_lock**: allows multiple threads concurrent read access to a resource but grants write access to only a single thread when all current readers release the resource. The first thread to request write access to a resource that is currently locked by any number of readers is guaranteed to obtain the resource as soon as all the current read locks on that resource are released. Once a writer has obtained the resource lock, all subsequent threads requesting read or write access will pend with no guarantee of which thread will obtain the resource when the writer releases the lock. The

last reader releasing a lock will awaken a pending writer. A writer will awaken all threads pending for the resource after releasing the write lock.

- **no_lock**: no attempt is made to lock a resource with this lock type. This lock type is used when it is known that the resource is guaranteed to be protected by some other mechanism and locking would be redundant. For instance, the POQ locking mechanism that arbitrates which protocol requests can execute concurrently, provides a course grained locking scheme that could be used to lock whole classes of resources, thus making the finer grained RDB locking mechanism redundant. Cursors are examples of resources that are locked via the POQ mechanism, thus, locking individual resources of type RT_CURSOR would be redundant.

All core resource types have a predefined lock type (see FIGURE 29). Extension resource types will specify the desired lock type at creation time (extension types are created via the *CreateNewResourceType* method of the RDB Monitor).

Core Resource Type	Lock Type
Window	reader_writer_lock
Pixmap	exclusive_lock
GC	exclusive_lock
Font	no_lock
Cursor	no_lock
Colormap	no_lock
Colormap Entry	no_lock
Other Client	no_lock
Passive Grab	no_lock

FIGURE 29 Core resource lock types.

9.3.2 Resource Lock Modes

While the resource lock type specifies the locking mechanism to be used for all resources of a particular type, the resource lock mode indicates the type of access an individual resource will require during the execution of a protocol request (resources are expected to be locked prior to beginning the execution of a protocol request). The lookup/lock methods of the RDB Monitor require that a lock mode be specified for each individual resource.The following resource lock modes are available:

- read_lock_mode: a resource is required for read access. This mode can be specified for a resource of any lock type.
- write_lock_mode: a resource is required for write access. This mode can be specified for a resource of any lock type.

Note that requesting either read_lock_mode or write_lock_mode for a resource with an exclusive lock type will result in the resource being locked exclusively. Also, requesting either mode for a resource with a no_lock type will have no effect.

The resource lock mode has no dependency on the resource lock type. This is required because some resources are only known to be members of a certain class at the time

they are to be located in the RDB and locked. A resource class may consist of several different resource types that use different locking schemes. The access mode required for a resource should always be known regardless of whether the resource is known to be of a particular type or merely a member of a resource class.

9.3.3 Locking Precedence

To avoid deadlock situations when locking multiple resources for use by a protocol request, a locking precedence based on resource type and resource id has been defined. The following steps define the proper resource locking order.

- Each resource type has a unique numeric value. Resources should first be ordered for locking in ascending order based on the resource type.
- Resources of the same type should then be ordered by ascending order of resource id.

Core resource types have the lowest values and are listed in numeric order in FIGURE 29. The values of extension resource types begin at a value greater than the highest core resource type and are assigned in increasing order at creation time (*CreateNewResourceType* method of the RDB Monitor). Thus,

- Core resources are always locked prior to extension resources.
- Extensions can control the locking order of their resource types by arranging the calls made to *CreateNewResourceType* in the desired locking order.

For performance reasons, the RDB Monitor does not enforce this locking precedence, although the *LockResources* method does provide proper ordering as an option.

The RDB Monitor provides locking methods for individual resources or a group of resources in one operation. If resources are locked individually it is the responsibility of the implementor to submit locking requests in the proper order. If resources are locked as a group, the implementor has the option of arranging the resources in the proper order before calling into the monitor or having the monitor determine the proper locking order.

Resource classes complicate this process. Since a resource class may include several resource types, the actual type of the resource cannot be known until it is located in the RDB and the type extracted. Thus, the locking order cannot be determined until some of the resources have been looked up. The ordering algorithm for the *LockResources* method, if requested, will assure that all classes are resolved prior to ordering on resource types. Again, for performance reasons this is left as an option for the implementor. The implementation of the sample MTX server never requires the RDB to perform the ordering algorithm.

NOTE: For some classes, it may be known that the order of locking for resources in that class could not cause a deadlock situation. This is the case with the only core resource class, RC_DRAWABLE. It contains window and pixmap resource types. It is known that a thread will never pend when trying to obtain a window lock inside the RDB. So it does not matter whether a window is locked before a pixmap or vise versa. Thus, all resources of this class only need to be ordered by resource id, the type is irrelevent. In

the case of RC_DRAWABLE, the resource can be locked in sequence with other resources without having to lookup the resource first and determine the type. Avoiding the lookup may provide a performance advantage. Again, the *LockResources* method will assure that a list of resources known by type and/or class will be locked in the proper order if the ordering option is requested. If an implementor is certain that the order of resources provided to the RDB are currently in an acceptable locking order, the extra overhead of locking can be avoided.

9.4 External Interface

9.4.1 Add or Free Resources

These routines are responsible for linking new resources into the RDB and removing existing resources from the RDB. X resources are always created outside the RDB and then added to it. The following access methods allow for adding resources to and removing resources from the RDB.

Bool AddResource(XID, RESTYPE, pointer);

void FreeResource(XID, RESTYPE);

void FreeResourceByType(XID, RESTYPE, Bool);

void FreeClientResources(ClientPtr);

void FreeAllResources();

9.4.2 Locking Resources

9.4.2.1 Lock Resource Individually (Generic Interface)

A generic interface is provided by the RDB that allows the locking and unlocking of any type of resource, core or extension. A resource can be locked given a resource id and a resource type/class.

pointer LockResourceByType(XID, RESTYPE, AccessMode);

pointer LockResourceByClass(XID, RESTYPE, AccessMode);

void UnlockResource(XID, RESTYPE, pointer);

9.4.2.2 Lock Resource Group (Generic Interface)

A generic interface is provided by the RDB that allows the locking and unlocking of a group of core and/or extension resources. Information about the resources is supplied to these routines by the caller via a resource array. Each resource array element contains the following information; a resource id, a resource type or class, an access mode, and storage for a return pointer.

These routines accept a boolean value that indicates whether the supplied resources should be ordered in the proper locking precedence before attempting any locks. If this option is not selected, the resource array must already be ordered for the proper locking precedence by the caller. Generic pointers for each resource are returned in the resource array. If any bad resource ids are specified, nothing is locked and an error status is returned. These lock and unlock routines have to be called in pairs with the same

resource array. Once the resources have been locked, the resource array should not be altered until after the unlock.

int LockResources(ResArrayPtr, int numRes, Bool doOrdering);

void UnlockResources(ResArrayPtr, int numRes, Bool doOrdering);

9.4.2.3 Lock Core Resources

A set of special case lookup/lock and unlock routines are provided for all the combination of resource lookups required by the core server. These are provided for performance improvements.

int LockDrawableAndGC();

int LockTwoDrawablesAndGC();

int LockDrawable();

int LockPixmap();

int LockGC();

int LockTwoGCs();

int LockWindow();

int LockTwoWindows();

int LockColormap();

int LockFont();

int LockCursor();

void UnlockDrawableAndGC();

void UnlockGC();

void UnlockTwoGCs();

void UnlockWindow();

void UnlockTwoWindows();

void UnlockColormap();

void UnlockFont();

void UnlockCursor();

void UnlockDrawable();

void UnlockPixmap();

9.4.3 Extend Type or Class

These routines allow for new resource type and resource classes to be defined. Associated with a new resource type is a pointer to a function to call on deallocation of the resource, the resource lock type to be associated with this type of resource, a boolean specifying whether this resource is a common resource, and an integer specifying the byte offset in the resource data of the resource lock record.

RESTYPE CreateNewResourceType(DeleteType, LockType, Bool, int)

RESTYPE CreateNewResourceClass();

9.4.4 Initialization and Termination

These routines are called to initialize and cleanup the RDB monitor. Initialization of extensions that require creation of new resource types and/or classes can be much more efficient if it is known that the thread performing the initialization is the only existing thread. When the monitor is first initialized, it assumes it is in single thread mode and does not attempt to grab locks when new resources types and or classes are created. A special interface routine exists that is called by the MST prior to creating new threads that informs the RDB monitor that it must now start obtaining locks to modify the resource type and/or class information. Also, as new clients connect with the server, the RDB client structures (hash tables, caches, etc) are allocated.

```
void RDBInitializeMonitor();
void RDBCleanupMonitor();
void RDBEnterMultiThreadMode();
```

9.4.5 Miscellaneous

The following miscellaneous routines are part of the RDB monitor. *FakeClientID* allocates an unique resource id to be assigned to a newly created resource. Many resource ids are allocated by the client and included as part of a protocol request that creates the resource. But for those resources that do not have a resource id supplied by the client, this routine will allocate a unique one. *LegalNewResourceID* will verify that a resource id is valid, currently unallocated in the RDB, and associated with a given client. *ChangeResourceValue* allows the resource data currently associated with a given resource id and type to be changed. *ShareResource* is called whenever a thread expressess passive interest in a common resource. This routine will merely increment the reference count associated with the common resource, while protected by an appropriate RDB lock.

```
XID FakeClientID(int);
Bool LegalNewResourceID(XID, ClientPtr);
Bool ChangeResourceValue(XID, RESTYPE, pointer);
void ShareResource(RESTYPE, pointer, pointer);
```

9.5 Internal Organization and Implementation

9.5.1 Locking Mechanisms

There are two classes of objects that need to be locked in the RDB Monitor; the internal database structures and the individual resources. The locking mechanisms of the RDB attempt to allow maximum concurrency inside the monitor without adding additional overhead for the most common monitor operations. This section will present the internal components of the RDB Monitor and the locking implementations, and conclude with a discussion of the performance tradeoffs incurred by this design.

9.5.1.1 Locking RDB Internal Structures

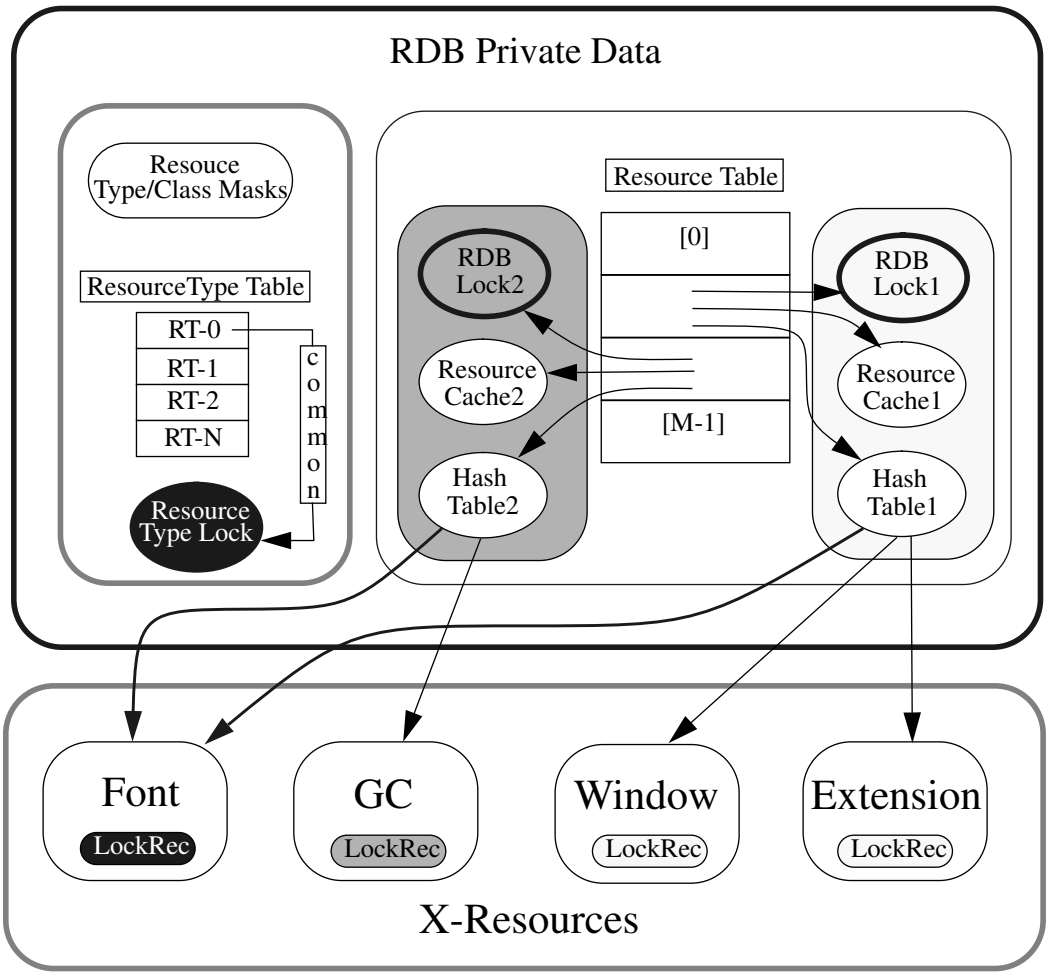


FIGURE 30 RDB locking mechanism.

FIGURE 30 shows the internal structure of the RDB database and the relationship between the RDB private data and the X resources. There are three locks shown in this diagram, each with a different shading. The shading extends over the data objects that the lock protects. For example, RDB Lock2 is a lock associated with element 2 of the resource table. It protects the resource cache, hash table, and hash lists (see FIGURE 31) associated with element 2 of the resource table. It also protects the lock record of any resource contained on one of the protected hash lists that is not a common resource. For instance, the GC lock record is protected by RDB Lock2 because it is not a common resource. The font is a common resource and is not protected by RDB Lock2 because there may exist other pointers to it from hash tables not protected by this lock (hash table 1 in the example).

Resource Table: The resource table is the top level structure of the RDB. It is a simple array of elements that is indexed by client id. The id of the client that created the

resource will be included in the resource id and is extracted as a key into the table. The table contains one entry for each possible client, 0 to MAXCLIENTS-1. Each element of this table contains a pointer to an RDB lock, a pointer to a hash table, a pointer to a resource lookup cache, and various other status information. The data items for ResourceTable[*n*] are created when client *n* connects to the server.

RDB Lock: Each entry in the ResourceTable has an associated RDB Lock. This lock consists of a mutex, a condition, and a boolean. The mutex is held while manipulating any of the private data items associated with that resource table entry. It will also be held while modifying the lock record associated with a non-common resource that appears in the RDB resource lists of the associated hash table. The condition variable is used to block a thread that attempts to modify a lock record for a resource that is already locked. In this case, the mutex is released and the thread waits on the condition variable until the resource is unlocked and the thread is signaled. The boolean is true if any thread is waiting on the condition variable and indicates a condition signal is possibly required when a resource proteced by this lock is released.

Resource Cache: Each entry in the Resource Table has an associated resource cache. The resource cache is an optimization that improves lookup performance by caching the last accessed resource id and the pointer to that resource for a given resource type. The cache is searched first before performing the hashing algorithm and traversing the hash lists during a resource lookup.

Hash Table: Each entry in the Resource Table has an associated hash table. The hash table is an array of pointers to lists of RDB resource nodes (see FIGURE 31). Any RDB resource node that appears on a list in the hash table is only accessible through this hash table. Thus, if the hash table is locked, all the resource nodes accessible from that hash table are locked.

RDB Resource Lists: Each entry in any RDB hash table contains a pointer to a list (potentially empty) of RDB resource nodes (see FIGURE 28). All RDB resource nodes on the list are only accessible though the hash table in which the list is anchored.

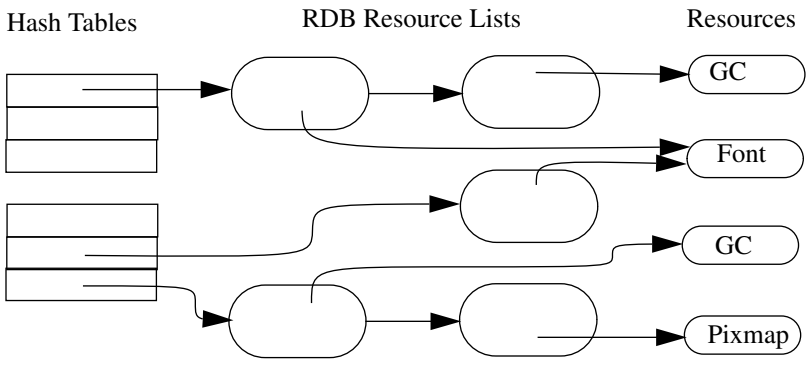


FIGURE 31 RDB Resource Lists.

Resource Lock Record: Each resource will contain a resource lock record as part of the resource data. See Section 9.5.1.2 for details on the lock record and the resource locking mechanisms.

Resource Type Table: The resource type table is an array of elements indexed by resource type. This array grows as new resource types are created. Each element contains the following information about a particular resource type; a pointer to a delete function to be called when a resource of this type is freed, the lock type, byte offset of the lock record within the resource data, and a pointer to an RDB lock record if the resource type is a common resource (NULL if not a common resource).

The RDB lock associated with common resource types must be held when trying to lock a common resource. For example, in FIGURE 30, the font lock record must be protected by the RDB lock in the ResourseType table. The RDB lock associated with the ResourceTable (shaded in gray) must be held to locate the resource via the cache or hash table. Once located, the lock must be released and the new lock in the ResourceType table (shaded in black) obtained to allow modification of the resource lock record. Again, only common resources have to switch locks between resource lookup and resource locking.

The ResourceType Table is accessed during nearly all RDB operations. It is accessed to determine the lock type and lock record offset during lock/unlock operations. For performance reasons, these accesses should be fast and not require any additional locking. When a lookup occurs, an RDB lock from the Resource Table will be held. Holding this lock will also guard against modifications to the ResourceType Table. To guarantee this, all Resource Table RDB locks must be obtained before modifying the ResourceType Table. It is assumed that this requirement is reasonable because the need for modifying this table will be rare. The table is only modified when new resource types are created,. This normally occurs during extension initialization. An optimization will exist that will avoid this overhead if the MST is the only executing thread (initialization of static extensions occur in the context of the MST). Dynamically loadable extensions will pay the overhead when they are initialized, if they create new resource types outside the context of the MST.

9.5.1.2 Locking Individual Resources

Resources are locked individually in the RDB monitor, providing a fine grained locking scheme. The locking process involves locating a resource in the database followed by an exclusive access and modification of the resource lock record. Once the resource lock record has been modified, it is locked for the requesting access. The locking required to locate the resource in the database is discussed in Section 9.5.1.1.

Depending on the final implementation, the lock record may be a structure or merely a 32 bit word. Contained in the lock record will be a 32 bit word referred to as the **resource lockbits**. The purpose of the lockbits field is to show the active state of lock access to the resource.

The lockbits field is divided as shown in FIGURE 32.

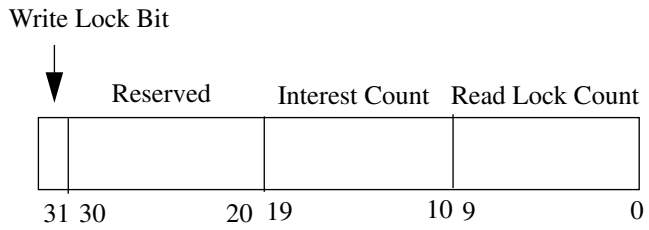


FIGURE 32 Resource lockbits format.

The lockbits field can only be accessed while being protected by an exclusive RDB lock as discussed in Section 9.5.1.1. Both the exclusive resource lock and the readers/writers resource lock are implemented on top of the lockbits.

Write Lock Bit: This bit is ON when a resource is locked exclusively or a thread has attempted write access on readers/writers lock. When the write lock bit is ON, no other read or write accesses are permitted for this resource.

Read Lock Count: This bitfield is an indication of the number of concurrent read accesses being made on this resource. The read lock count is only required in the implementation of a readers/writers lock. Write access cannot be granted to a resource until its read lock count becomes zero.

Interest Count: This bitfield is an indication of the number of accesses that are blocked pending a state change in the lockbits. This is used for resource deletion, and as an optimization when some thread has located a resource in the RDB and was blocked in its attempt to lock it. Access will be granted to this resource when it becomes available but requires that the resource is not deleted while the thread is pending. A resource can be removed from the RDB with a non-zero interest count, but the memory cannot be deallocated until the entire lockbits field becomes zero. A thread attempting to free the resource memory will block until the lockbits field becomes zero. Also, the interest bits can be used to indicate whether a condition broadcast is required after unlocking a resource. A broadcast is only needed if a thread is pending on this resource, and the interest bits will be greater than zero if this is true.

The following are the locking algorithms for various types of access. Note that an RDB lock must be held prior to executing any of these algorithms and that the condition wait step will cause the lock to be released before pending and reobtained before resuming.

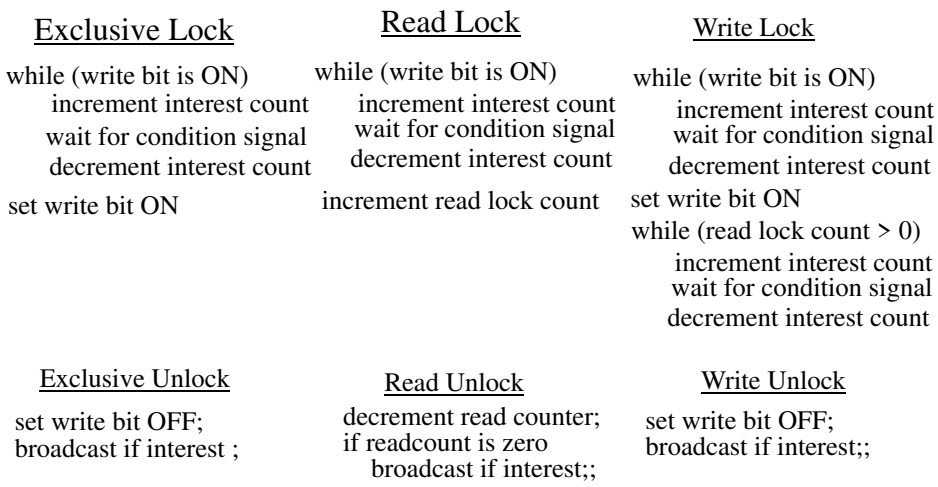


FIGURE 33 Resource lock and unlock algorithms.

9.5.2 Performance Considerations

9.5.2.1 Locking Analysis

A goal for the RDB monitor is to allow for a high level of concurrency without requiring high locking overhead. The performance of the locking mechanisms described in Section 9.5.1 is analyzed along with an alternative locking mechanism with an explanation as to why the alternative was not selected for use by the RDB monitor.

The most common operations in the monitor are lookup/lock and unlock, and should be optimized. Ideally, multiple RDB operations from different clients each requiring *n* resources would:

- execute concurrently
- not contend for the same mutexes
- require the overhead of only one mutex lock and unlock per operation
- hold the mutex for a short period of time
- only awaken pending threads that can resume execution on unlock
- minimize the number of condition signals required on unlock

Both methods mentioned below assume the use of the resource lockbits mechanism for implicitly locking individual resources (see Section 9.5.1.2). The mutex that protects the lockbits and the condition queues that threads wait on are the issues considered in this section. The benefits of the implicit locking mechanism include the fact that mutexes are not held on a resource for the amount of time the resource requires locking (the duration of a protocol request) and, in many cases, multiple resources can be locked under the protection of a single mutex. It is assumed that resources likely to be referenced in a

group can be organized such that the same mutex protects the lockbits of all in the group. The method described in Section 9.5.1.1 will protect all non-common resources created by the same client under one mutex. It is expected that, in general, when a group of resources are required by a protocol request, that they will all have been created by the same client, the client that is making the request.

9.5.2.1.1 Single mutex method

This section discusses a locking mechanism utilizing a single exclusive mutex with multiple associated conditions. The mutex would be used to protect all private data and the lock records for all resources in the RDB. Using a single mutex would ensure that the first two bullet items would not be met.

If a single condition was associated with this mutex, then all threads waiting for any resource in the RDB would line up on a single condition queue requiring a broadcast when any resource that has a thread pending for it is unlocked. This could result in a thundering herd of threads resuming execution that merely get placed back on the condition queue, but pay the context switch overhead. To avoid this, multiple conditions would be used.

So how many conditions should be used and how should they be divided?

One solution would be place a condition variable in each resource's lock record along with a waiting flag. It would be known that all pending threads on a condition queue would be waiting for a specific resource. For exclusive lock types, pending threads could be awakened one at a time in a round robin fashion. For readers/writers lock types, the last reader to release a resource could be guaranteed that the first thread in the condition queue is the writer who should receive this resource next, thus a condition signal will awaken this thread and allow the others to remain pending. Writers releasing a resource would want to broadcast and awaken all threads waiting for that resource.

One condition per X-resource would be too expensive in terms of thread resources. Another solutions would be to associate a condition and wait flag with each element of the resource table (the same thing as the RDB lock mechanism described in Section 9.5.1.1 without the mutex). This would result in shorter condition queues where each thread on a particular queue is known to be waiting for a resource owned by the client associated with that resource table entry. However, each pending thread on a condition queue would have to be awakened when any resource owned by that queues client is released. An advantage is that this may result in fewer pthread condition calls when releasing a group of resources, the broadcast can wait until all resources owned by a particular client have been released.

Although this method seems to meet many of the listed goals, it was not chosen because it restricts concurrency in the RDB monitor and results in a highly contended mutex. It is believed that both concurrency and significant reductions of mutex contention inside the RDB can be achieved without trading off much of the benefits of this method. NOTE: This method does have appeal because it is simple and easily solves some locking problems in the monitor, locking of shared resources for example. Since all RDB operations are very short in duration, it is still questionable that a single mutex would become highly contended and the loss of concurrency would be significant enough to justify the tradeoff.

9.5.2.1.2 Multiple RDB lock method

This is the method described in Section 9.5.1. It was selected because it was felt that it would provide a high level of concurrency without much locking overhead. The following observation is used to support the selection of this method.

> The resource requirements of a protocol request from a particular client will usually consist of a few resources that were created by that client and are not accessed by another client.

Since the resource table elements are indexed by client id (usually extracted from the resource id), it is expected that the RDB lock associated with that entry will be uncontended most of the time. Usually, only the client who created the resource has interest in it. This would provide high concurrency. Also, since groups of resources required by a request usually are created by the same client, one mutex would have to be obtained to lock the whole group of resources.

Given the assumption, this method would meet many of the same performance goals that the single lock method provided, but allow higher concurrency and less contention on the mutexes. All the goals listed in Section 9.5.2.1 are met with the exception that on unlock, threads not waiting on the resource being unlocked will be awakened if some other thread is waiting for that resource. Since all threads waiting for any resource owned by a particular client line up on the same condition queue, all must be awakened when a resource is unlocked and has threads waiting for it.

Disadvantages of this method are that, access of common resources will incur extra overhead, and creation of new types and/or classes outside the MST will be expensive (see Section 9.5.1.1).

9.5.2.2 Readers/Writers Lock

The readers/writers lock algorithm is designed such that when a writer requests a resource lock, it is guaranteed to get that lock as soon as all the current read locks held on that resource are released. That is, any subsequent read or write requests on that resource will block behind the writer. This will avoid potential starvation of a writer thread. However, it will ensure that the thread that just released a read lock on that resource will *not* be able to reobtain this resource in the next request, if needed, and will have to block, giving up the remainder of its timeslice. NOTE: None of the core resources ever request write access for a resource protected by a readers/writers lock. This lock type is provided as a feature for extensions.

9.5.2.3 Expanded Interface for Core Resources

Since the lookup/lock and unlock operation are performance critical, the interface to the RDB monitor was expanded to include special case routines for locking and unlocking core resources. By special casing these lookup routines, pieces of information that would have to be retrieved by the generic interfaces is known merely by context. Avoiding these retrievals and the overhead of loading and unloading a resource array improves lookup performance up to 20% as compared to going through the generic interface. A 20% improvement on lookup performance may or may not significantly affect the overall performance of a protocol request. Short protocol requests where the lookup time is a significant portion of the entire operation will benefit. The generic

interface will be used for looking up non-core resource types and possibly a group of resources with a mixture of core and non-core resource types.

CHAPTER 10 Pending Operation Queue Monitor

10.1 Overview

The Pending Operation Queue (POQ) is a complex lock which provides MTX with a mechanism to lock requests, groups of resources, screen real estate, windows and window subtrees. The purpose of the POQ is to provide a flexible locking mechanism by which threads can publicize server operations to other threads. Server operations are defined to be protocol requests from CITs and the processing of device input from DITs. A server thread can look in this queue and determine if there will be a lock conflict with any other thread.

10.1.1 POQ Concepts

Every server operation is required to access the POQ monitor. Information is added to the monitor at the beginning of each server operation. This information is then checked against other information in the monitor to determine if there is a lock conflict with another currently executing server operation. If a conflict exists and cannot be avoided, the POQ will block that server operation until the conflicting operation has completed.

The POQ has provided solutions to some very difficult problems, such as locking pixels on hardware which supports dumb frame buffers, software cursors, locking subtrees of the window hierarchy, GrabServer, and others.

Features of the POQ are:

- Provides an efficient mechanism for acquiring multiple locks.
- Multiple locks can be acquired easily without possibility of deadlock.
- Provides sequencing of server operations to prevent one thread from being starved out by other threads contending for the same locks. The POQ is implemented as a priority queue to enforce this ordering for conflicting server operations.
- Variable lock granularity. Fine grained locking of windows and window subtrees is permitted. It also allows locking of screen real estate by detecting region conflicts. In addition, coarse grained locking of cursors and colormaps is enforced.
- Easily extensible allowing server extensions to access features of the POQ.

10.2 Data Objects

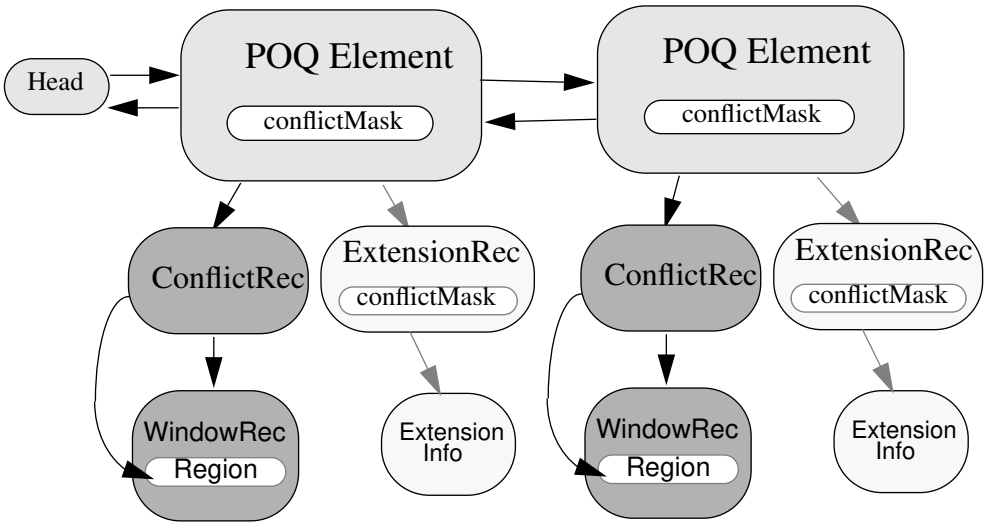


FIGURE 34 POQ Data structures

Data objects utilized by the POQ monitor are as follows:

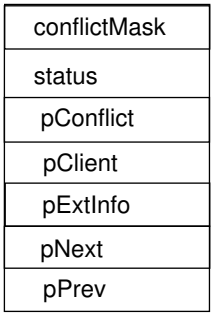
- POQ Element.
- GrabServer Element.
- ConflictRec.
- ExtensionRec.
- WindowRec.
- ConflictMask Bits.

10.2.1 POQ Element

POQ elements are private within the POQ monitor. There is one element for each potential CIT and DIT on the system. All POQ elements are allocated at server initialization time. This array of [0..MAXCLIENTS+NUMDITS] is pre-allocated in order to improve virtual memory locality, since these structures are small and will be referenced often. A free list of elements is not required since each thread can have at most one server operation executing at a time.

A POQ element is assigned to a CIT or DIT at thread initialization time. For CITs, the POQ element is referenced using the client id. For DITs, indexes start at MAXCLIENTS and go to the number of DITs on the system.

POQ element



The components of the POQ element are as follows:

conflictMask: (CM) 32 bit entity describing the locks to be acquired for a particular server operation. Exclusive locks are represented by a single bit in the *conflictMask*. Read/Write locks require two bits (one bit for read, the other for write).

status: this mask indicates the current status of a thread. Status can indicate that the thread is either BLOCKED, RUNNING, or WAITING. The BLOCKED and RUNNING bits are set by the thread itself to indicate whether that thread is runnable on the POQ or blocked waiting on another thread. If a thread is blocked, it will set the WAITING bit of the thread which it is blocked on. This notifies the blocking thread that another thread is waiting for a lock it holds. The blocking thread should check at some convenient location and decide whether or not to release locks to the waiting thread (this is what is referred to as coming up for air).

pConflict: points to a ConflictRec provided by the server operation. This structure is used to pass objects into the monitor to be used in conflict avoidance.

pClient: points to the client owning this POQ element. If the owning thread is a DIT, this points to the dummy clientRec allocated by the DIT.

pExtInfo: points to an ExtensionRec if this operation was initiated by a server extension request (as opposed to a core request). The ExtensionRec structure contains information about the server extension issuing the request.

pNext and **pPrev:** points to the next and previous elements on the POQ.

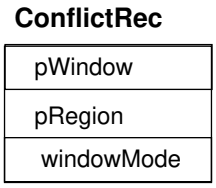
10.2.2 Grab Server Element

The GrabServer element is a specially allocated POQ element that is used to free up a CIT's POQ element when an active server grab is in progress. This element will conflict with all other requests and is placed at the head of the POQ when a GrabServer request becomes active. All requests inserted behind it in the POQ will block. Requests issued from the grabbing CIT will insert its POQ elements at the tail of the GrabServer queue, instead of the POQ. Since the DIT is not affected by a server grab, it will also insert its element at the tail of the GrabServer queue and contend with requests from the grabbing CIT. The GrabServer element is private within the POQ monitor. This is described in more detail in section 10.3.4.

10.2.3 ConflictRec

The POQ monitor is able to make use of server objects when deciding if a lock conflict on the POQ can be avoided. The ConflictRec provides a mechanism to pass these objects into the monitor. For the core server, these objects consists of *windows* and *regions*. When the POQ detects a window hierarchy, geometry, or render conflict, it will use these objects to determine whether or not the conflict can be avoided.

Every operation that requires the POQ to perform conflict avoidance will set the appropriate fields of the ConflictRec before locking the POQ. For each POQ element, there exists exactly one ConflictRec. The ConflictRec is external to the POQ monitor.



The components of the ConflictRec are as follows:

pWindow: indicates the window or window subtree that is locked.

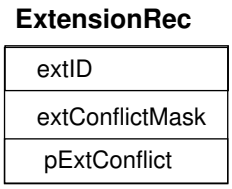
pRegion: indicates the region of the screen to lock during a render operation. The region depends on the server operation to be performed.

windowMode: indicates whether *pWindow* refers to a single window or the highest window in a window subtree.

10.2.4 ExtensionRec

The POQ monitor provides a mechanism for server extensions to use the features of the POQ monitor when locking extension operations. Each extension is provided with its own set of conflict bits and a mechanism to register objects in the monitor to be used in conflict avoidance. There also exists a function which will register a conflict avoidance routine defined by the extension writer. This conflict avoidance routine is called from within the POQ monitor when an extension conflict is encountered.

There may exist at most one ExtensionRec for each POQ element. The ExtensionRec is external to the POQ monitor.



The components of the ExtensionRec are as follows:

extID: uniquely identifies the extension issuing the request.

extConflictMask: indicates the locks required by the extension operation. This is in addition to any of the core server locks required.

pExtConflict: used by the extension writer to pass objects into the monitor to be used in conflict avoidance. This points to a structure defined by the extension writer. The contents of the structure must be such that the extension conflict avoidance routine will know how to decipher it. The POQ monitor does not need to know the details of this structure.

10.2.5 WindowRec

Because of the high degree of contention for windows and the frequency with which the window hierarchy is traversed, windows in MTX are locked at a very fine level using the POQ. Individual fields of the WindowRec are protected by multiple locks on the POQ. These locks and the window fields they protect are described in detail in section 10.2.6.

In the RDB monitor (see CHAPTER 8), windows are only locked for read access (they are never write locked). This is done to prevent the window from being deleted until the POQ monitor locks the portion of it that is of interest to the operation.

10.2.6 Conflict Mask bits

The following list describes the conflict bits used in the core MTX server. The first list contains all of the Read/Write bit pairs. The second list contains exclusive Write only bits. It is important to realize that fields contained within the same 32 bit word (for example, two shorts) must be protected by the same lock. If this is not the case, each field should be moved into its own 32 bit word. The actual bit location is implementation dependent.

10.2.6.1 Read/Write Conflict Mask Bits

- CM_R_HIERARCHY
- CM_W_HIERARCHY

Setting these bits will read or write protect a window subtree hierarchy. Conflicts are avoided by determining that the conflicting windows are not in the same window subtree. Therefore, the window placed in the ConflictRec should be the highest window in the subtree which requires a lock. The following fields in the WindowRec are locked:

```
WindowPtr       parent;
WindowPtr       nextSib;
WindowPtr       prevSib;
WindowPtr       firstChild;
WindowPtr       lastChild;
```

- CM_R_GEOMETRY
- CM_W_GEOMETRY

Setting these bits will read or write protect a window's geometry attributes. Conflict avoidance is done by comparing window pointers in the ConflictRec. If the windows are different, no conflict exists. All render operations in MTX require that the CM_X_RENDER and CM_R_GEOMETRY bits are set in the conflict mask. The following fields of the WindowRec are locked:

```
DrawableRec     drawable;
RegionRec       clipList;
RegionRec       borderClip;
DevUnion        *devPrivates;
union_Validate  *valdata;
RegionRec       winSize;
RegionRec       borderSize;
DDXPointRec     origin;
unsigned short  borderWidth;
PixUnion        background;
PixUnion        border;
pointer         backStorage;
unsigned        backgroundState:2;
```

```
unsigned        borderIsPixel:1;
unsigned        backingStore:2;
unsigned        saveUnder:1;
unsigned        DIXsaveUnder:1;
unsigned        bitGravity:4;
unsigned        winGravity:4;
unsigned        overrideRedirect:1;
unsigned        visibility:2;
unsigned        mapped:1;
unsigned        realized:1;
unsigned        viewable:1;
```

This lock also protects the following fields of the WindowOptRec:

```
VisualID        visual;
unsigned long   backingBitPlanes;
unsigned long   backingPixel;
RegionPtr       boundingShape;
RegionPtr       clipShape;
```

- CM_R_COLORMAP
- CM_W_COLORMAP

Setting these bits will allow read or write access to all colormap and colorMapEntry objects in the server. This will also read or write protect the window's attributes related to colormaps. The following fields of all WindowOptRec's are locked:

```
Colormap        colormap;
```

- CM_R_EVENT_PROP
- CM_W_EVENT_PROP

Setting these bits will read or write protect all window's attributes related to event propagation. The following fields of the WindowRec are locked:

```
unsigned short  deliverableEvents;
Mask            eventMask;
unsigned        dontPropagate; /* used to be 3 bits */
```

This lock also protects the following fields of the WindowOptRec:

```
Mask                dontPropagateMask;
Mask                otherEventMasks;
struct _OtherClients    *otherClients;
struct _GrabRec         *passiveGrabs;
struct _OtherInputMasks *inputMasks;
```

- CM_R_SCREENSAVER
- CM_W_SCREENSAVER

Setting these bits will read or write protect access to the following screensaver data:

ScreenSaverTime
ScreenSaverInterval
ScreenSaverAllowExposures
screenIsSaved
savedScreenInfo

- CM_R_FONTPATH
- CM_W_FONTPATH

Setting these bits will allow read or write access to all font path data

10.2.6.2 Exclusive Conflict Mask Bits

- CM_X_SERVER

Setting this bit will lock access control, host data, and extension initialization data.

- CM_X_ICCCM

Setting this bit will lock access to the atom database, selections, and window properties. The following fields of all WindowOptRec's are locked:

```
PropertyPtr        userProps;
```

- CM_X_FONT

Setting this bit will lock access to all font related data.

- CM_X_RENDER

Setting this bit will write protect pixels in the frame buffer. The region specified in the ConflictRec will be used when avoiding a render conflict. All render operations in MTX require that the CM_X_RENDER and CM_R_GEOMETRY bits are set in the conflict mask.

- CM_X_CURSOR

  Setting this bit will write protect all cursor objects in the system. The window's attributes related to cursors will also be locked. The following fields of the WindowOptRec are locked:

```
unsigned        cursorIsNone; /* used to be 1 bit */
CursorPtr       cursor;
```

10.2.6.3 Misc. Conflict Mask Bits

- CM_NO_CONFLICTS

  Setting this bit will indicate that the operation has no conflicts with any other operation except for grab server. CM_NO_CONFLICTS and CM_GRABSERVER are logically a read/write pair.

- CM_GRABSERVER

  Setting this bit will indicate that the request wishes to grab the server. This bit will conflict with every other bit in the mask including CM_NO_CONFLICTS.

- CM_ EXTENSION

  Setting this bit will indicate that this request originated from a server extension. In case of a conflict here, the POQ monitor will call the extension conflict avoidance function for this extension to determine if the conflict can be avoided.

10.3 Concurrency Issues

10.3.1 POQ Monitor Internals

The internal elements of the POQ monitor are protected by an exclusive global Mutex:

```
xmutex_t        POQMutex;
xcondition_t    POQCondition;
Bool            POQWaiting;
```

The POQCondition variable is used to inform other threads that the state of the POQ has changed. The POQWaiting flag is set when some object or operation protected by the POQ is not available and a thread will have to wait. The POQWaiting flag is an optimization so that a broadcast is not required every time an element is removed from the queue, only when there is a thread waiting.

The POQ monitor executes within a context of the MST, a CIT or a DIT. The MST will initialize the POQ monitor, including the POQ mutex and condition variable, the GrabServer element, and the conflict avoidance mechanism. The CIT and DIT will access the POQ for every server operation executed.

10.3.2 POQ Access

Access to the monitor is through the POQLock() and POQUnlock() function calls. The POQLock() function will block a thread when a collision is detected and cannot be avoided. The POQ determines which threads are eligible to run and which threads are blocked. A general algorithm used when processing a server operation is as follows:

- Set conflict bits.
- Set up the ConflictRec.
- POQLock().
- Execute the server operation to be performed.
- POQUnlock().

The POQLock() and POQUnlock() functions are described below:

POQLock()

- Lock the POQMutex.
- If the server is grabbed and the client is the grabbing client or the DIT, insert the POQ element at the tail of the GrabServer queue. Otherwise, insert the POQ element at the tail of the POQ.
- Traverse the queue from the current element to the head and compare conflict masks of each element on the queue to the current. This operation consists of a bit comparison of the two conflict masks. (if (current->conflictMask & next->conflictMask))
- If a conflict is detected, determine if the conflict can be avoided by using the information contained in the ConflictRec.
- If the conflict cannot be avoided, set the POQWaiting flag, and wait on the POQ condition for the conflicting operation to complete. Repeat from step #3 until no conflicts exist. If no conflict was detected, continue.
- Unlock the POQMutex.

POQUnlock()

- Lock the POQMutex.
- Remove the element from the POQ.
- If the POQWaiting flag is set, broadcast the POQCondition and clear the POQWaiting flag. This tells any waiting threads that an entry in the POQ has been removed.
- Unlock the POQMutex.

Although the POQ may be highly contented, we expect that under normal server loads, the POQ will have few elements on the queue at any given time and even fewer that actually collide.

The diagram in FIGURE 38 illustrates how several conflicting server operations are processed on the POQ using the above algorithm. The shaded blocks represent operations that are blocked.

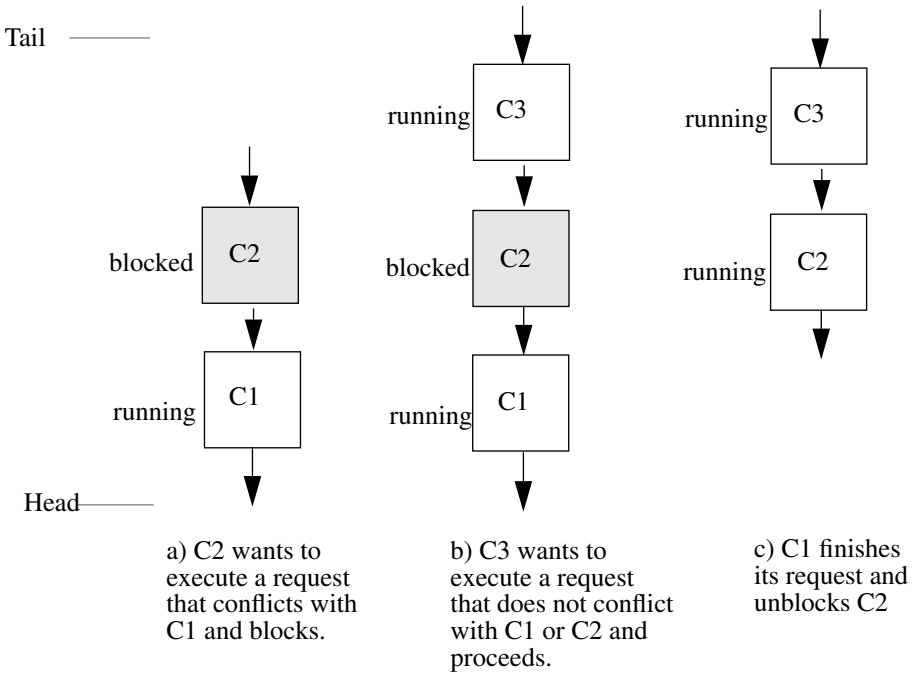


FIGURE 35 Processing requests on the POQ

10.3.3 Conflict Detection/Avoidance

10.3.3.1 Window Conflicts

A request which requires a CM_R_HIERARCHY, CM_W_HIERARCHY, CM_R_GEOMETRY, or a CM_W_GEOMETRY lock must also set the *pWindow* and *windowMode* fields in the ConflictRec. *pWindow* indicates the window or window subtree to lock. *windowMode* indicates whether *pWindow* refers to a single window or the highest window in a window subtree. When a window hierarchy or geometry conflict occurs, the POQ monitor will use this information to determine if the conflicting requests can proceed in parallel.

If *WindowMode* indicates that *pWindow* refers to a single window, then lock conflict avoidance is done by comparing the two conflicting window pointers to see if they are the same. If they are, then the thread performing the comparison is blocked.

If *WindowMode* indicates that *pWindow* refers to a window subtree, then lock conflict avoidance is done by determining if either window is a descendant of the other. If they are, then the thread which initiated the comparison is blocked. This is explained in greater detail in the next section.

10.3.3.2 Window Subtree Locking

This optimization applies only to operations which require locking of the window hierarchy. These operations manipulate either single windows or windows contained in arbitrary subtrees. The goal is to provide a mechanism which provides as much concurrency as possible while incurring low cost in conflict detection overhead.

It is difficult to determine whether or not arbitrary subtrees intersect. By increasing the granularity so that only top-down(TD) subtrees are used, we can simplify intersection detection while sacrificing only a small amount of concurrency. A TD subtree is defined as a tree from a given window, down to and including all its leaf nodes. The root of a TD subtree is the top most affected window.

To minimize the amount of tree walking done when determining if subtrees intersect, a level number will be added to each window. This level will indicate the height of that window in the tree. For example, the root window's level will always be 0, all direct descendants of the root will be level 1, and so on (see diagram below). As the tree is modified, the level numbers in each window will need to reflect any changes made. This may penalize some requests, such as ReparentWindow, that need to update the level numbers of every window in the subtree.

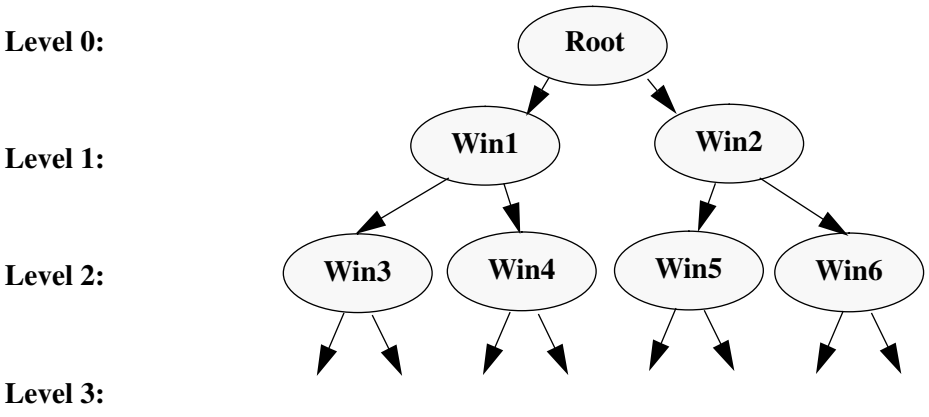


FIGURE 36 Window hierarchy showing level numbers.

To determine if two TD subtrees intersect:

- If both root windows of the TD subtree are at the same level and are not the same window, the subtrees do not intersect.
- Start with the window with the higher level number.
- If the lower window is a single window, the subtrees do not intersect.
- Check to see that the lower window is not an ancestor of the upper window by walking back the parent pointers and comparing window ids. Stop at the level number of the lower window.
- If the lower window is not an ancestor, the subtrees do not intersect.

The following algorithm describes the steps the POQ will perform when determining window and window subtree conflicts. The window mode is used to indicate whether the conflict window is the root of a subtree or a single window.

```
DetermineWindowSubtreeConflict()
{
        get requests conflict window (REQcw)
        get requests window mode (REQmode)

        set windowConflict to FALSE

        for (each conflicting element in the POQ)
        {
                get POQ conflict window (POQcw)
                get POQ window mode (POQmode)

                if (level(REQcw) == level(POQcw))
                {
                        if (REQcw == POQcw)
                                set windowConflict to TRUE
                }
                else if ((level(REQcw) < level(POQcw)) and
                        (REQmode == SUBTREE))
                {
                        if (REQcw is an ancestor of POQcw)
                                set windowConflict to TRUE
                }
                else if ((level(REQcw) > level(POQcw)) and
                        (POQmode == SUBRTEE))
                {
                        if (POQcw is an ancestor of REQcw)
                                set windowConflict to TRUE
                }
        }
}
```

10.3.3.3 Region Conflicts

A request which requires a CM_X_RENDER lock must also set the *pRegion* field in the ConflictRec. This is used to indicate the region of the screen to lock during a render operation. This region depends on the server operation to be performed. Normally, it is one of the regions contained in the WindowRec, either the *clipList*, *borderClip*, *winSize*, or *borderSize*, although it may be a custom region (for example, the region of the screen affected by a ConfigureWindow request). When a render conflict occurs, the POQ monitor will use this region to determine whether the conflicting requests can proceed in parallel. On systems using packed pixel dumb frame buffers, the regions are expanded in the x direction so that they are word aligned. This is done to prevent multiple processors from accessing pixels contained in the same word in memory simultaneously (although they may be in different regions). The expanded regions are then checked for intersection. In FIGURE 38, P1 and P2 are independent processors which both wish to modify pixels contained in the same word in the frame buffer at the same time.

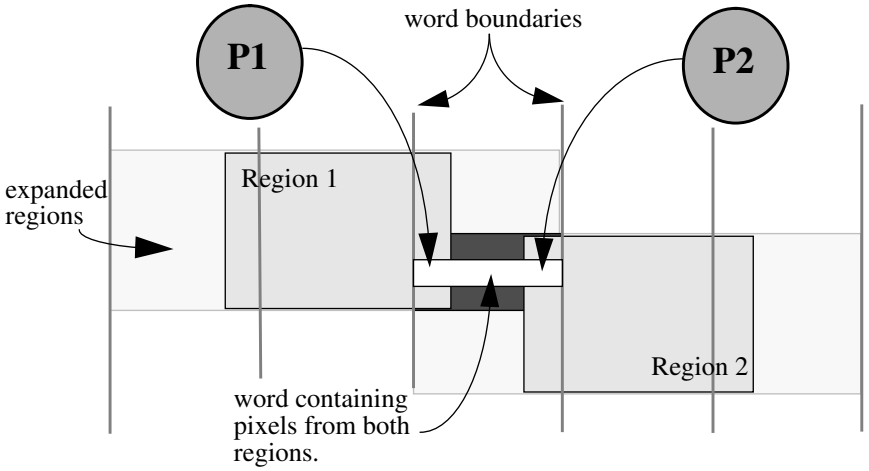


FIGURE 37 Expanding regions for packed pixel frame buffers.

10.3.3.4 Extension Conflicts

In addition, the POQ monitor provides a way for extensions to create additional conflict bits that are private to each extension. A request which sets the CM_EXTENSION_BIT must also set the *extConflictMask* field and the *pExtConflict* field in the ExtensionRec. The *extConflictMask* field contains an indication of the locks needed for this extension operation. These locks are defined by the extension writer. The *pExtConflict* field in the ExtensionRec points to a ConflictRec defined by the extension writer. The POQ monitor knows nothing about the internals of this structure. It merely passes it along to the extensions conflict avoidance function when an extension conflict occurs. Conflict resolution for extension operations is handled by the POQ monitor in the following way:

- Check the POQ for conflicts with core server operations.
- If a CM_EXTENSION_BIT conflict is encountered and both conflicts originated from same extension (extID's are the same), then check for extension conflicts using the extConflictMask, otherwise continue.
- If an extension conflict is detected, call the conflict avoidance function for this extension.
- If the conflict avoidance function returns TRUE, then block this thread until the conflicting operation completes, else continue.

10.3.4 Grab Server

The GrabServer and UngrabServer protocol requests are processed in the following way:

POQGrabServer()

- Lock POQMutex.
- Add the grabbing client's POQ element to the tail of the POQ.
- Increment the GrabServer counter.
- Wait until the client's POQ element is at the head of the POQ.
- Save grabbing client's id.
- Swap the GrabServer element for the client's POQ element.
- Unlock POQMutex.

POQUngrabServer()

- Lock POQMutex.
- Decrement GrabServer counter.
- Remove GrabServer element from POQ.
- Broadcast to any waiting threads.
- Unlock POQMutex.

The GrabServer request will pend until it gets to the head of the POQ. When the grabbing client's element reaches the head of the POQ, it is replaced with the GrabServer element. This is done to block other requests behind it in the POQ while freeing up the grabbing clients element for use in future requests. As long as the server is grabbed, the grabbing client will insert his element at the tail of the GrabServer queue instead of the POQ. Since the DIT is not affected by a GrabServer request, it will need to contend with the grabbing client's POQ element. This requires that operations from the DIT are also inserted at the tail of the GrabServer queue.

Protocol requests from clients who aren't the grabbing client, are inserted normally at the tail of the POQ. Since the GrabServer element conflicts with every server request, all other requests will block on the GrabServer element until the UnGrabServer request is processed.

In FIGURE 38, each POQ element is represented by a box. Shaded boxes indicate requests that are blocked. Since each CIT can only have one server operation executing at any one moment, the client that grabs the server may have at most one POQ element. So, the GrabServer POQ element is used as a place holder for the grabbing client (as well as the DIT) to insert its privileged operations on the POQ and be assured of immediate execution.

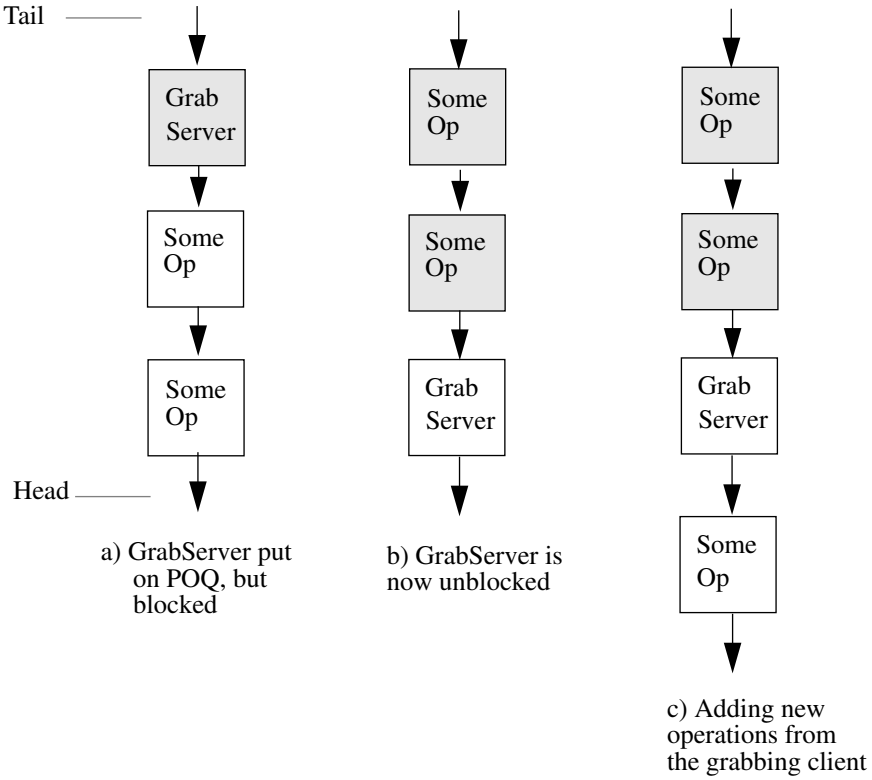


FIGURE 38 GrabServer example

10.4 External Interface

The POQ monitor is accessed using the following interface:

10.4.1 Initialization and Termination

These routines are used to initialize and destroy the POQ monitor. They are called from the MST during server initialization and shutdown.

- POQInitializeMonitor ();
- POQDestroyMonitor ();

The following function is provided to tell the POQ what type of hardware is available on each screen. This will allow the POQ to adjust region calculations accordingly when detecting region conflicts. This function should be called from the DDX function *InitOutput()* after a new screen has been added.

- POQSelectRegionConflictType (screenNum, hardwareType);

where hardwareType is one of the following:

POQ_1_BIT_PER_PIXEL
POQ_8_BITS_PER_PIXEL
POQ_16_BITS_PER_PIXEL
POQ_24_BITS_PER_PIXEL
POQ_32_BITS_PER_PIXEL
POQ_USE_GRAPHICS_ACCEL

This routine is called from a CIT when a new client is created. It assigns a POQ element to a client and performs any client specific initialization in the POQ monitor.

- POQInitClient (pClient);

This routine is called from the MST when a server extension initializes. This will register a conflict avoidance function for this server extension.

- POQSetExtConflictFunc (extID, *ExtConflictFunc());

10.4.2 Lock/Unlock

These routines are used by server operations when accessing the POQ monitor.

- POQLock (pClient, conflictMask);
- POQUnlock (pClient);

10.4.3 Come Up For Air

The following function causes a client's POQ element to be removed from its current location in the POQ, and placed back at the tail. This effectively allows threads which are blocked waiting for this client to proceed. This function is intended to be called from ddx render routines which may take some indefinitely long period of time to execute (such as PEX render requests). The calling function must be in a position to guarantee that anything affected by unlocking objects protected by the POQ will be reset before continuing the request. For example, if a render request gives up the CM_R_GEOMETRY lock on a window, it must check the window's origin and clipList before continuing that render operation.

- POQComeUpForAir (pClient);

10.4.4 Grab Server

These routines are used to grab and ungrab the server. They are called from a CIT issuing a GrabServer or UngrabServer protocol request.

- POQGrabServer (pClient);
- POQUngrabServer (pClient);

10.4.5 Convenience Macros

A set of convenience macros is also provided to help set up the ConflictRec. These are referenced from outside the POQ monitor before the POQ is locked.

- POQ_SET_NULL_CONFLICT (pClient);
- POQ_SET_ALL_CONFLICT (pClient);
- POQ_SET_RENDER_CONFLICT (pClient, pDrawable, pGC);
- POQ_SET_WINDOW_CONFLICT (pClient, pWindow, regionType);
- POQ_SET_DRAWABLE_CONFLICT (pClient, pDrawable);
- POQ_SET_REGION_CONFLICT (pClient, pWindow, pRegion);

Macros are also provided to set up the POQ for extension operations:

- POQ_SET_EXTENSION_CONFLICT (pClient, extConflictMask, pExtConflict);

CHAPTER 11 Device/Event Monitor

11.1 Overview

The Device/Event Monitor (DEM) will arbitrate access to the device and event related data objects in the server. These objects include such things as the keyboard and pointer device records, focus records, and sprite information. FIGURE 39shows a high level view of the DEM and the various categories of routines available in the external interface.

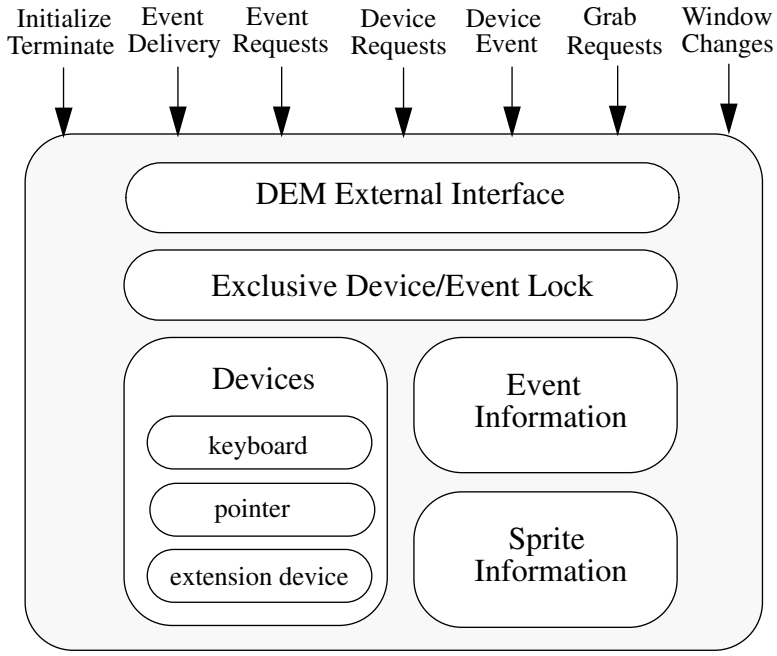


FIGURE 39 Device/Event Monitor.

11.2 Data Objects

FIGURE 39 shows the main data objects protected by the DEM; devices, event data, and sprite data. All of this data is protected by a single exclusive lock.

Devices: Structures for all devices initialized by the sample MTX server will be allocated in the DEM and protected by its locking mechanism. A keyboard and a pointer device is created during initialization of the server. Additional devices can be created and managed under the DEM via extension interfaces.

Event Information: Event data includes objects such as the event queue for frozen devices, event reply data, and event propagation data.

Sprite Information: Sprite data includes objects such as the currently displayed cursor, physical and logical region constraints for the cursor hot spot, physical and logical position of the cursor hot spot, the window containing the cursor, and a trace buffer of all windows on the path from the cursor window to the root. The sprite trace buffer is recalculated every time a pointer motion event causes the cursor to enter a new window. Handles for the cursor and window resources are the protected by the DEM, but the actual resource data is locked separately.

11.3 Concurrency Issues

The DEM executes within the context of a CIT, DIT, or MST. When one of these threads calls a routine that needs to access a device or event related object, the routine will obtain the DE exclusive lock. These routines are then considered the interface to the DEM. Some of these routines are Proc<request> level routines, while others are lower level routines (see Section 11.4). Note that POQ locks must be held prior to the DE lock being obtained.

There is additional event related information in the window record that is not protected by the DEM, but is protected by the POQ. The CM_R_EVENT_PROP and CM_W_-EVENT_PROP conflict masks are used to protect these fields (see Pending Operation Queue Monitor for details).

The course grained locking scheme used by the DEM is considered to be reasonable because the monitor is accessed infrequently and for short periods of time. The following assumptions support the conclusion that a single exclusive mutex will not be highly contended and thus provide a reasonable locking scheme.

- Device mapping and control requests are very infrequent.
- Grabs requests are very infrequent.
- Event playback requests are infrequent.
- Interactive device input occurs infrequently relative to other server operations.
- The window manager will generate a high proportion of the non-device events.
- Processing of window manager events will occur serially because they occur within the context of a single CIT.

11.4 External Interface

The DEM consists of the device and event data objects and the locking mechanism for these objects. This section will list all core server routines that access the device/event data and require locking. Extension routines that access this data will use the locking routines provided by the DEM and will be considered to be a part of the monitor. Note that many of these routines access data objects outside the DEM and will require additional locking. Typically, these objects are windows and cursors and are locked via the POQ.

11.4.1 Data Locking

The following routines are called from within the monitor to implement the exclusive locking mechanism required for device/event data objects. These routines do *not* have to be called when using any of the other routines listed in the external interface. They are not currently called from outside the monitor.

- LockDeviceEvents();
- UnlockDeviceEvents();

11.4.2 Initialization and Termination

- DEMInitializeMonitor();
- DEMCleanupMonitor();
- InitAndStartDevices();
- NumMotionEvents();
- QueryMinMaxKeycodes();
- DefineInitialRootWindow();
- InitEvents();
- CloseDownDevices();
- ReleaseActiveGrabs();

11.4.3 Event Delivery

These routines are used to deliver events generated outside the DEM. For instance, expose events are generated outside the monitor and delivery of event to the proper client will require manipulation of the device/event structures. These routines will acquire the device/event lock, perform the required function, and release the lock.

- TryClientEvents();
- MaybeDeliverEvents()
- DeliverEvents();

11.4.4 Event Requests

These routines implement top level protocol requests that manipulate the event structures in some way.

- ProcAllowEvents();
- ProcSendEvents();
- ProcGetMotionEvents();

11.4.5 Device Requests

These routines implement top level protocol requests that manipulate the device structures in some way. For instance, key maps can be read or changed, device control structures can be modified, etc.

- ProcGetModifierMapping();
- ProcSetModifierMapping();
- ProcGetKeyboardMapping();
- ProcChangeKeyboardMapping();
- ProcGetPointerMapping();
- ProcSetPointerMapping();
- ProcChangeKeyboardControl();
- ProcGetKeyboardControl();
- ProcChangePointerControl();

- ProcGetPointerControl();
- ProcQueryKeymap();
- ProcBell();
- ProcWarpPointer();

11.4.6 Device Event

These routines process input events from the core devices.

- ProcessKeyboardEvent();
- ProcessPointerEvent();

11.4.7 Grab Requests

The following routines activate/deactivate device grabs, or create/remove passive grabs on a device element. These are all top level protocol request implementations.

- ProcGrabButton();
- ProcGrabKey();
- ProcGrabKeyboard();
- ProcGrabPointer();
- ProcUngrabButton();
- ProcUngrabKey();
- ProcUngrabKeyboard();
- ProcUngrabPointer();

11.4.8 Window Changes

The DEM maintains sprite information which includes a trace of all windows on the path from the window that contains the sprite to the root. When the hierarchy of the window tree changes in any way, the *WindowsRestructured* routine is called to recalculate this trace buffer. When a cursor is changed for a window, *WindowHasNewCursor* is called and may change the sprite information and redisplay the cursor if needed. See Section 11.5.1 for POQ locking requirements of calling routines.

- WindowHasNewCursor();
- WindowsRestructured();
- DeleteWindowFromAnyEvent();
- EventSelectForWindow();
- EventSuppressForWindow();
- RecalculateDeliverableEvents();

11.5 Internal Organization and Implementation

The DEM consists merely of device and event related data objects and a locking mechanism that protects these objects with a single exclusive mutex. Any routines that require

access to the data objects are considered to be part of the monitor. Conceptually, it helps to think of the DEM as a monitor, although it is not actually implemented like the other MTX monitors with private data and an external interface all in one module. Any extension routines that access these data objects, the xinput extension for example, will be required to use the same locking mechanism and be considered part of the monitor.

11.5.1 Locking Requirements

The DE exclusive lock is obtained after the POQ locks have been obtained. This lock is released prior to any attempts to obtain additional MTX server locks.

Many of the routines in the DEM are top level protocol request implementations, such as *ProcSendEvents*. These routines lock all required RDB resources, then the POQ, and finally the DE objects. The xinput extension will also contain top level protocol request implementations that will be considered part of the DEM.

Other routines in the DEM are lower level routines called during the execution of a protocol request, such as *DeliverEvents*. These routines obtain the DE lock and perform the requested operation. All required RDB resource locks and POQ locks must be held prior to calling one of these functions. FIGURE 40 shows a list of routines and their POQ locking requirements. Protocol requests that call these routines will obtain these locks, along with any additional POQ locks, prior to calling the DEM.

DEM Routine	POQ Requirements
TryClientEvnets	none
MaybeDeliverEvents	r_event_prop
DeliverEvents	r_hierarchy, r_geometry, r_event_prop, x_cursor
ProcessKeyboardEvents	r_hierarchy, r_geometry, r_event_prop, x_cursor
ProcessPointerEvents	r_hierarchy, r_geometry, r_event_prop, x_cursor
WindowHasNewCursor	r_event_prop, x_cursor
WindowsRestructured	r_event_prop, x_cursor
DeleteWindowFromAnyEvent	r_hierarchy, r_geometry, r_event_prop, x_cursor
EventSelectForWindow	r_hierarchy, w_event_prop
EventSuppressForWindow	r_hierarchy, w_event_prop
RecalculateDeliverableEvents	r_hierarchy, r_event_prop

FIGURE 40 POQ requirements of lower level DEM routines.

11.5.2 Device/Event Access Routines

FIGURE 41 shows the routines that are included in the monitor and how the monitor is referenced from other parts of the server. Most of the DEM access routines are inside the bubble labeled *Device Event Data Access*. All routines in the server that access the device event data objects are included here. These routines perform no locking, the locking is performed prior to calling. The diagram shows the different areas where DE locking occurs.

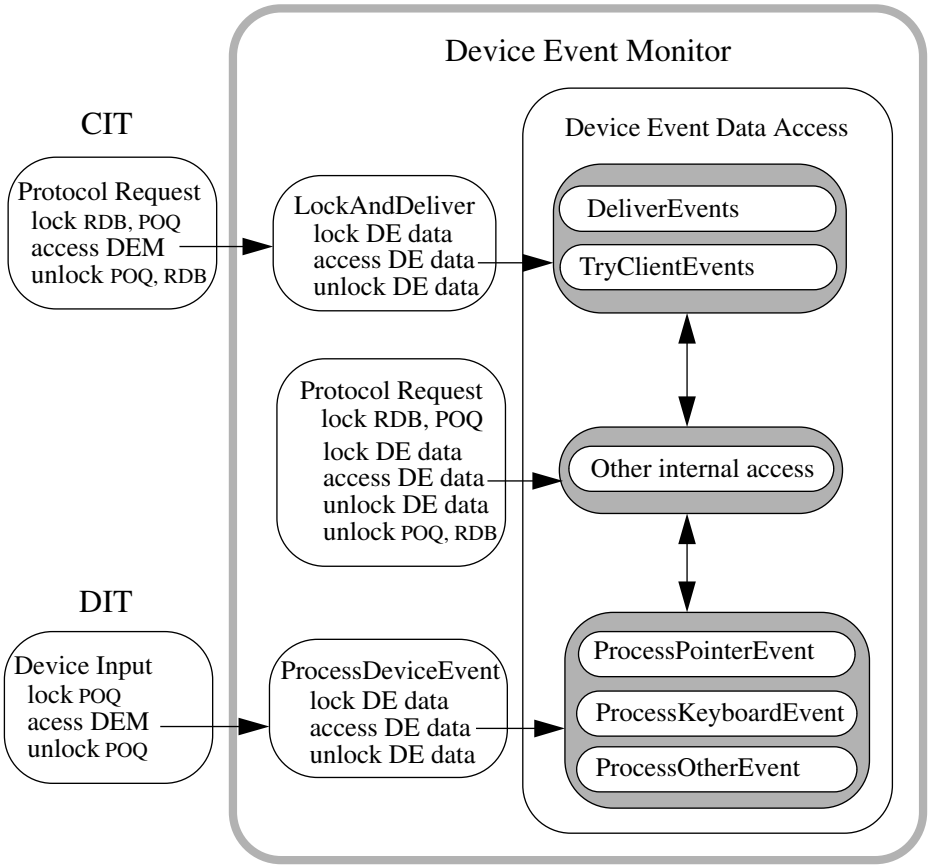


FIGURE 41 Device Event Monitor Internals.

11.5.3 Implementation Alternative

An alternate implementation for locking the device event data is via the POQ with either a single exclusive bit in the conflict mask, or possibly, read/write bits in the conflict mask. This may actually have some advantages over the current implementation because requests that generate several events in separate calls to *LockAndDeliverEvents* would not require grabbing and releasing the device/event mutex on each call. This method would be a little coarser grained locking scheme but still a simple implementation.

These objects could also have been locked with a finer grained scheme. The device and event data could be separated and protected by different locks, and potentially, each device could be locked individually. However, the extra complexity of finer grained locking was judged to not be a reasonable tradeoff given the infrequency of the access to these data items.

CHAPTER 12 Message Output Monitor

12.1 Overview

The Message Output Monitor (MOM) manages all output communications from the MTX server to any client. Any data that flows from the server to a client must pass through the MOM.

The server will send replies, events, error, and connection information to a client. Replies, events, and errors will be referred to as messages. The message is the basic unit around which the MOM components are built.

FIGURE 42 shows a high level view of the MOM and the various categories of routines available in the external interface. The diagram also shows that all Client Output Threads are created by this monitor.

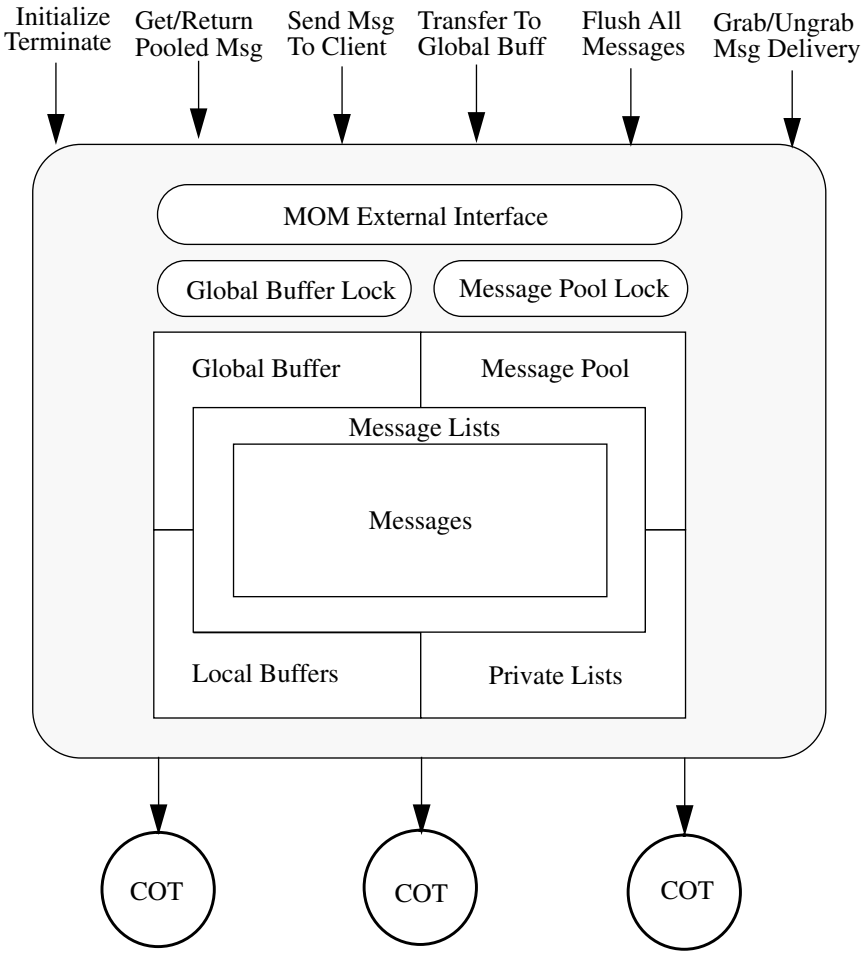


FIGURE 42 Message Output Monitor.

12.2 Data Objects

As FIGURE 42 shows, the core data object of the MOM is the message. Individual messages are linked together to create message lists which are the components of the main data structures, the message pool and message buffers.

Messages: The server will send replies, events, and errors to clients. These data items are generically referred to as messages. A reply message may include two pieces, a fixed portion and an optional data portion. The fixed portion will always be contained in a message while the optional data portion will be attached . A message will contain storage for the actual data to be sent to a client (the fixed portion for replies), a message type field, a pointer to the optional reply data, and a message pointer that allows for message

lists to be created. Note that the data portion of a reply, if any, must be allocated and initialized outside the monitor and attached to the message containing the fixed portion of the reply.

Message Lists: Individual messages are linked together to create message lists. A message list may be empty or contain one or more messages.

Message Pool: All messages that are not in use will appear in the message pool. The message pool maintains a set of message lists and allocates messages for private use by a requesting thread. The message pool will grow and shrink to accommodate varying server loads. The number of messages allocated by the MOM is only limited by the amount of memory the server can obtain, and will be reduced as messages are returned to the pool and the total pool size exceeds a predefined maximum.

Message Buffers: A message buffer contains a single message list for each of MAXCLIENTS potential target clients. The *global message buffer* contains lists of messages that are deliverable to target clients. There is one global message buffer in the monitor. A *local message buffer* is allocated for every thread that may potentially generate messages. During the initialization of such a thread, the monitor is instructed to allocate a local message buffer and associate it with that thread. Local message buffers contain lists of messages that have been generated for target clients by a server operation but are not deliverable until all messages to be generated by that operation have been completed.

The MOM is designed and implemented such that data movement from buffer to buffer does not occur. Message storage is obtained from the monitor and the contents generated directly into this message storage. Moving messages between the message pool, local message buffer, global message buffer, and back to the pool merely involves pointer shuffling.

12.3 Concurrency Issues

12.3.1 Message States and Transitions

A message may be in any one of four states inside the MOM. The memory for a message is allocated and managed by the monitor. Once created, a message is placed in the message pool and cycles through the various states inside the monitor until the server

determines the memory can be deallocated. FIGURE 43 shows the states a message may transition to during its lifecycle.

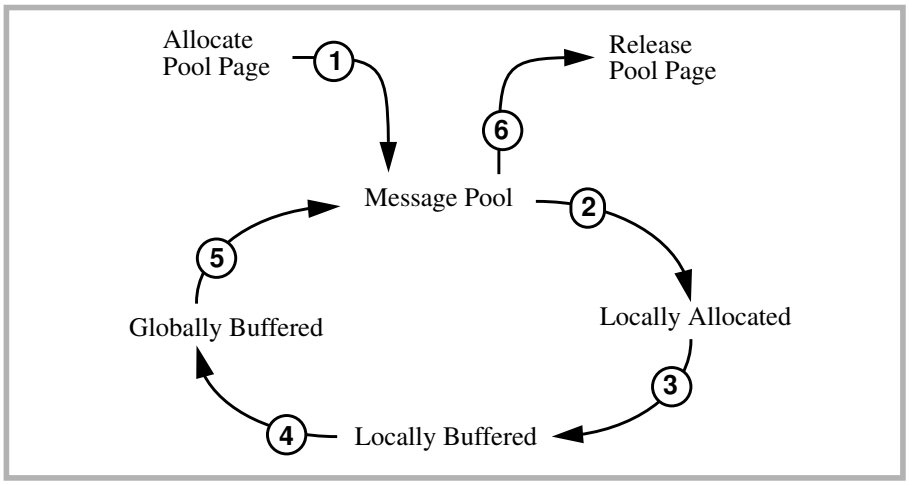


FIGURE 43 Message states and transitions.

Transition 1: A message has been requested from the MOM and there are no free messages in the pool. A new page of messages is allocated and linked into the message pool.

Transition 2: A server operation needs to generate messages for target clients. Typically, this operation would be the execution of a protocol request by a CIT or the processing of device input by a DIT. The monitor is instructed to remove messages from the message pool and make them available for private use by the server operation.

Transition 3: A message has been generated for a specific client but is not deliverable until all messages to be generated by this operation are completed. The monitor is instructed to place the message in the local message buffer associated with the thread performing the operation.

Transition 4: A server operation has completed. All messages generated by that operation are now deliverable. The monitor is instructed to transfer all locally buffered messages for this thread to the global message buffer.

Transition 5: A flush has been requested. The monitor is instructed to write all messages to their target clients in the order they appear in the global message buffer's message queues. When the data has been written, the messages are returned to the message pool.

Transition 6: A message was returned to its page in the message pool. Receiving this message caused the page to have no remaining messages outside the pool, making it available for deallocation if the number of pages in the pool exceed a high watermark.

12.3.2 X Protocol Output Requirements

The X-Window system protocol defines three conditions that must be met concerning the flow of output from the server to a client. The following are those conditions along with a definition [SG90].

1. Whether or not a server is implemented with internal concurrency, the overall effect must be as if individual requests are executed to completion in some serial order, and requests from a given connection must be executed in delivery order (that is, the total execution order is a shuffle of the individual streams).
2. The effect of any other cause that can generate multiple events must effectively generate and queue all required events indivisibly with respect to all other causes and requests.
3. For a request from a given client, any events destined for that client are caused by executing the request must be sent to client before any reply or error is sent.

execution of a request: includes validating all arguments, collecting all data for any reply, and generating and queuing all required events.

12.3.3 Meeting the X Protocol Output Requirements

12.3.3.1 Design Features

The following design features are used to verify that the MOM meets the requirements of the X-Window system protocol concerning the flow of output from the server to a client.

- Messages will be generated by a server operation only after locking all the shared resources required by the operation.
- A message is considered to have been generated when it is queued in a local message buffer.
- A message is considered to be queued and deliverable once it has been transferred to the global buffer.
- All messages generated by a single protocol request or by device input processing will be buffered locally by the thread performing the operation.
- Individual messages are added to a local buffer by appending them to the list of messages already generated by the same server operation destined for the same target client.
- All message lists in a local message buffer will be transferred to the global message buffer after all messages have been generated by a server operation and before the locked resources are released.
- Message lists are always appended to any existing list for the target client in the global message buffer.
- Messages will be delivered to a client in the order they appear in the global message buffer.
- The global message buffer is protected by a single exclusive mutex which is held during the transfer of all local message lists. Thus, the transfer is effectively atomic.
- Protocol requests from a given connection will execute serially in the delivery order.

12.3.3.2 Meeting Condition 1

> Whether or not a server is implemented with internal concurrency, the overall effect must be as if individual requests are executed to completion in some serial order, and requests from a given connection must be executed in delivery order (that is, the total execution order is a shuffle of the individual streams).

One requirement for this condition to be met is that all messages from a particular request must be delivered in a contiguous byte stream to the target client. Without this type of delivery, the effect that a request executes to completion could not be certain. There is no particular delivery order for this contiguous stream, with respect to requests from other connections. Since messages are delivered in the order they appear in the global buffer and all messages from a particular request are atomically transferred to the global buffer, the design assures that the overall effect is that request execute in *some* serial order.

Another requirement is that requests from a given connection be executed in delivery order. In the sample MTX server, requests from a given connection will execute serially in the delivery order.

12.3.3.3 Meeting Condition 2

> The effect of any other cause that can generate multiple events must effectively generate and queue all required events indivisibly with respect to all other causes and requests.

Messages from requests on different connections can be generated concurrently provided that the contents of the messages do not originate from shared resources. When this is true, messages are *effectively generated* indivisibly with respect to each other, even if generated concurrently.

If the generation of messages requires data from shared resources, the generation of these messages must occur serially. Since shared resources are locked at the beginning of a request and all messages for a request will be generated and queued prior to releasing the resource locks it is certain that all messages for this request are *effectively generated* indivisibly with respect to other requests.

All messages from a given cause are *effectively queued* indivisibly with respect to other causes since all messages from a given cause are atomically transferred into the global message buffer.

12.3.3.4 Meeting Condition 3

> For a request from a given client, any events destined for that client are caused by executing the request must be sent to client before any reply or error is sent.

An individual request is executed to completion serially and all requests that generate events are implemented such that the events are generated and buffered locally before and replies or errors. Since messages are always appended to message buffer lists and the messages are delivered in the order they appear in the global message buffer, the condition is met.

12.3.4 Message Delivery Threads

A thread requesting delivery of all queued messages must not be penalized by attempting to send messages to some client. For instance, a DIT should not block because it attempts to write a device event to some client whose socket buffer is full. Since the transport layer of the sample MTX server could potentially block a thread when it attempts to write data, the thread requesting a flush should not attempt to write the data. Instead, a new thread is created whose sole purpose is to perform the write via the transport layer, while the thread performing the flush continues execution. These new threads that are created merely to deliver messages to a target client are called Client Output Threads (COTs). In certain cases a CIT is allowed to write messages directly to a client, see Section 12.3.4.2 for details.

12.3.4.1 Client Output Thread

COTs are created during a flush request and are only created by the MOM. A COT will remove the list of messages pending for a specific client from the global message buffer, and write them to the target client via the MTX server transport layer. When the write completes, the list of messages is returned to the global message pool and the thread exits. Since messages must be delivered to clients in the order they appear in the global message buffer, multiple COTs with the same target client must be sequenced (see Section 12.5.4 for implementation details).

12.3.4.2 Client Input Thread

Typically, a CIT that initiates a flush will have messages pending for its associated client. That is, CIT *A* creates messages to be delivered to client *A* and eventually requests a flush. As an optimization, a CIT is allowed to write messages destined for its associated client directly from the CIT. It is assumed reasonable for a CIT to block and not receive more requests from its client if the client is not keeping up with the server and reading its messages. When the CIT is able to complete its write, it will resume execution of protocol requests. This special case is provided as an optimization to avoid thread creation and context switch overhead.

12.4 External Interface

12.4.1 Initialization and Termination

These interface routines include the initialization and termination routines called from the MST during server initialization and shutdown. Also included are register routines that must be called once by every CIT and DIT. They allocate the local message buffers required by threads that generate messages.

- InitializeMessageMonitor();
- CleanupMessageMonitor();
- RegisterLocalMessageBufferForThread();
- UnregisterLocalMessageBufferForThread(int, MsgBufferPtr);

12.4.2 Get/Return Message List to/from Pool

These interface routines get and return lists of messages from the global pool.

- GetPooledMessages(int);
- GetPooledReplyMessage(int, MessagePtr *);
- ReturnPooledMessages(MessagePtr);

The following are convenience macros.

- GetPooledMessage();
- ReturnPooledMessage(MessagePtr);
- GetReplyMessage(<reply-type>, MessagePtr *);
- GetReplyPointer(<reply-type>, MessagePtr);
- GetEventPointer(<reply-type>, MessagePtr);
- GetErrorPointer(<reply-type>, MessagePtr);

12.4.3 Send a Message to a Client

These interface routines locally buffer messages for a target client. Also included in this category are routines to send the initial connection information directly to a requesting client.

- SendReplyToClient(ClientPtr, MessagePtr);
- WriteEventsToClient(ClientPtr, int, xEvent *);
- SendErrorToClient(ClientPtr, int, int, XID, int);
- SendConnectionSetupInfo(ClientPtr, xConnSetupPrefix, char *, int);

12.4.4 Transfer Local Message Lists to Global Buffer

Messages are generated and queued locally during the processing of an individual protocol request. After all messages have been generated by a request, the locally buffered messages are transferred to the global buffer. This is the only routine that places messages into the global buffer.

- TransferLocalMessagesToGlobal(MsgBufferPtr);

12.4.5 Flush All Messages

This routine flushes all messages in the global message buffer to the target clients.

- FlushAllMessages(ClientPtr);

12.4.6 Grab Message Delivery for Client

In the case of a reply with a very large data block, it may not be possible, or desirable, to allocate enough memory to buffer the reply with the normal message handling mechanism. The following interface routines allow for obtaining exclusive access to the output transport mechanism and sending large replies directly to a client in smaller chunks. The implementation of the *XGetImageReq* request grabs message delivery for the target client if the resulting reply would be greater than a defined threshold. See Section 12.5.5 for details on the implementation of this mechanism.

- GrabClientMessageDelivery(ClientPtr);

- UngrabClientMessageDelivery(ClientPtr);
- WriteReplyDataToGrabbedClient(ClientPtr, pointer, int);

12.5 Internal Organization and Implementation

12.5.1 Message Structure

Messages include replies, events, and errors. Events and errors are always 32 bytes in length. Replies on the other hand, consist of 32 bytes potentially followed by more additional bytes.

A reply always begins with a fixed structure, most of which are 32 bytes in length although a few are larger. Optionally, a reply can have a block of associated data that is delivered immediately following the fixed structure (the structure contains information about how many bytes are in the entire reply).

The *GetWindowAttributesReply* is an example of a reply that requires more than 32 bytes in the fixed reply structure, it requires 44 bytes. The *QueryFontReply* is the largest fixed reply in the core server, requiring 60 bytes in the fixed structure. The *GetImageReply* consists of a 32 byte fixed structure immediately followed by *n* bytes that define the image. The MTX message structure requires special fields to represent the various types of replies that may be generated by the server.

The MTX message structure is shown in FIGURE 44..

MTX Message

Fixed Reply, Event or Error Data (32 bytes)
Message Type
Large Reply Pointer
Reply Data Pointer
Reply Data Length
Next Message Pointer
Pool Page Pointer

FIGURE 44 Message contents.

Reply, Event, or Error Data: This field is a 32 byte array intended for the reply, event, or error data. Replies that require more than 32 bytes cannot be stored in this field, instead, the byte length of the actual reply is stored here.

Message Type: The message contains a reply, event, or error.

Large Reply Pointer: When a reply is grater than 32 bytes it must be allocated separately and the pointer to it is stored in this field. In all other cases this field contains a NULL pointer. Users of the MOM interface routine *GetPooledReplyMessage* will get a pointer to the reply storage area regardless of whether it has to be allocated separately or not. The allocation and deallocation of large replies is completely transparent to the user of the MOM.

Reply Data Pointer: If a reply includes data beyond that contained in the fixed structure (stored in the reply data area or pointed to by the large reply pointer), a pointer to that data is stored in this field. Otherwise this field is NULL. Storage for this data is allocated outside the monitor and a pointer to it attached to the message. The monitor will free this storage when the message has been flushed.

Reply Data Length: The length of the data, if any, pointed to by the reply data pointer.

Next Message Pointer: A link to the next message in a message list.

Pool Page Pointer: A pointer to the page of messages from which this individual message is allocated. This value allows for quick returns to the global pool.

12.5.2 Global Message Pool

Unused messages are maintained in a global message pool within the MOM. Messages are removed from the pool when they are allocated to a thread that needs to generate a message. They are returned to the pool by the MOM following a flush.

Storage for the messages are allocated in pages of fixed number of bytes. The first few bytes of the page will be used for a page header and the rest divided into messages. The page header contains a pointer to the list of available messages in that page. Multiple pages are maintained in the global message pool and are linked together via the pool page header (see FIGURE 45). New pages will be allocated when a request for a message is made and there are no free messages in the pool. The storage for message pages will be deallocated when all messages from a particular page are present in the message pool and the total number of pages exceeds a predefined maximum.

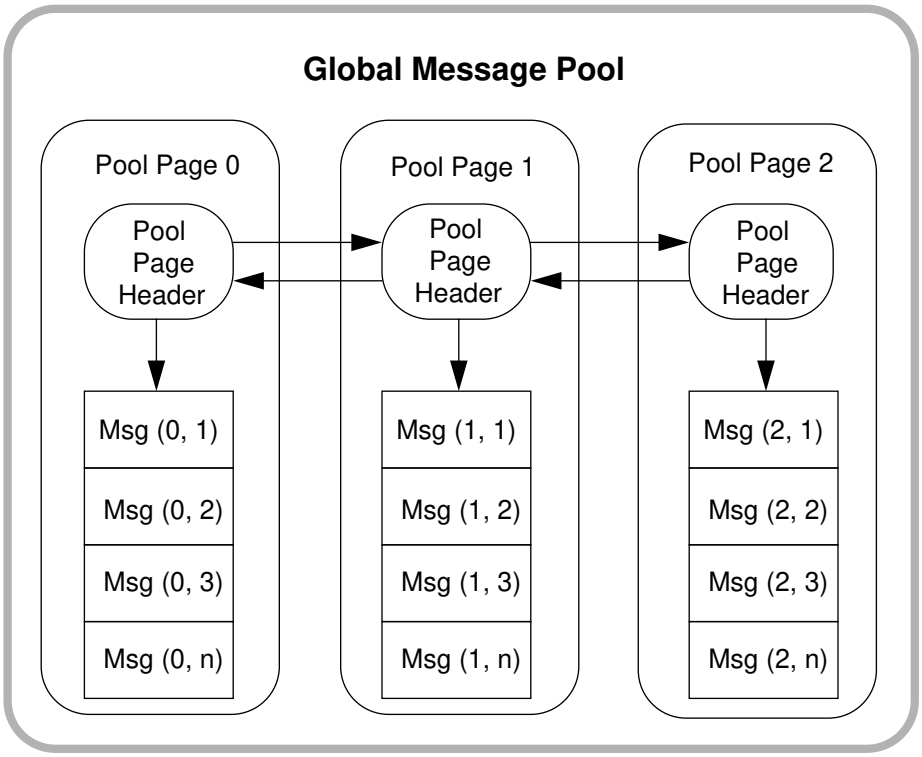


FIGURE 45 Global Message Pool.

If a request for a message cannot be met and a new page cannot be allocated (a malloc failure), the message pool will attempt to free up messages by flushing the global buffer. If this still does not free up a message, a search of the global message buffer will try to identify a client with an unreasonable number of messages pending and clobber that client, freeing up all its memory. If there are still no messages available, it will sleep for a few seconds and try the process over again. After a couple of retries, it will give up and return a NULL. Note that this is expected to be a rare condition.

12.5.3 Message Buffers

Once a message is generated by a server operation, it must be buffered until a flush request sends the data to the target client and returns the message structure to the message pool. Message buffers are the structures used to accommodate messages awaiting delivery.

The message buffer consists of a header followed by an array of structures, one structure for each of MAXCLIENTS potential target clients. Each element of the array will contain a pointer to the list of messages to be delivered to its target client and a pointer to the next array element that has at least one message pending. The header will contain a pointer to the first array element with at least one pending message (see FIGURE 46). The pending message linked list of a message buffer is used as an optimization that

allows for quick transfers of messages from local to global buffers and quick determination of which clients need to flush messages during a *FlushAllMessages* request.

The monitor contains a single global message buffer and multiple local message buffers, one local buffer for each CIT and DIT in existence. All message buffers, either local or global, have the same structure. The local message buffers are private to a thread and do not require any locking. The global message buffer is protected by a single exclusive lock.

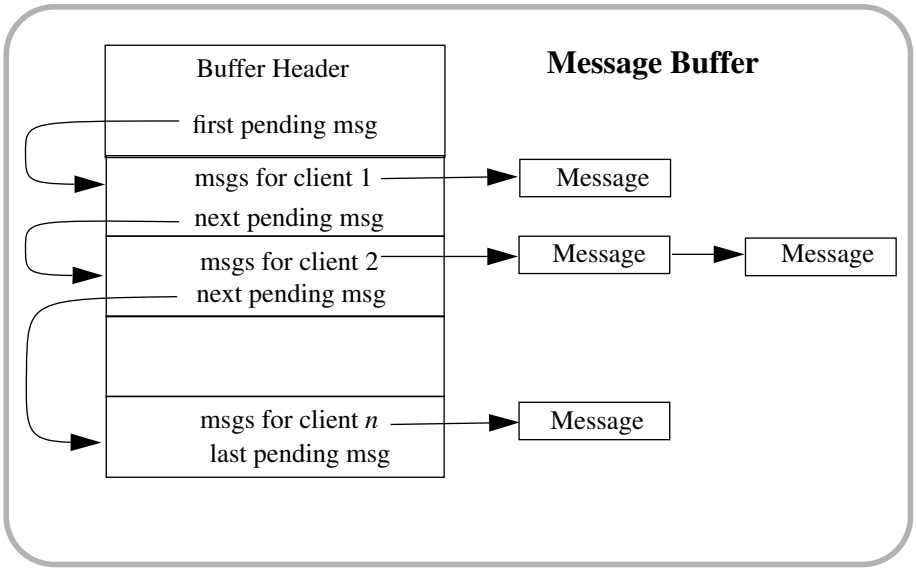


FIGURE 46 Message Buffer.

Messages are placed in a local message buffer by one of the *Send<message>ToCient* interface routines. Messages are removed from a local message buffer by the *TransferLocalMessagesToGlobal* interface routine.

Messages are placed in the global message buffer by the *TransferLocalMessagesToGlobal* interface routine. Messages are removed from the global buffer as a result of instructing the monitor to *FlushAllMessages* or *GrabClientMessageDelivery*.

12.5.4 Client Output Threads

Client Output Threads (COTs) are created solely to deliver messages to a target client. The COT will get the list of messages for its target client from the global buffer, setup an iov vector table and issue a blocking writev call to the file descriptor associated with the target client.

Messages must be delivered in the order they appear in the global message buffer. When a COT removes a list of messages from the global buffer it must make sure those messages are delivered before a subsequent COT delivers messages to the same client. To assure this, a COT will immediately obtain a lock associated with the target client and check to see if any other flush requests are pending for the same target client. The thread

will pend and wait to flush only if a flush is in progress and there are no other threads already pending for this target client. The message delivery algorithm is shown in FIGURE 47.

```
COT
{
     DeliverMessagesToClient(target client index);
     exit thread;
}


DeliverMessagesToClient(client index)
{
     obtain message delivery lock for target client;
     increment attempts-since-last-successful-flush counter;
     if any thread already waiting to flush to target client
     {
          release message delivery lock for target client;
          return;
     }
     while (thread actively flushing to target client)
     {
          wait on message delivery condition for target client;
     }
     set status for target client to actively flushing;
     release message delivery lock for target client;

     obtain global message buffer lock;
     get target client's message list from global buffer;
     release global message buffer lock;

     build iov vector table locally;
     writev(2);

     obtain message delivery lock for target client;
     set attempts-since-last-successful-flush to zero;
     set status for target client to not actively flushing;
     if (any threads waiting to flush to target client)
     {
          signal message delivery condition for target client;
     }
     release message delivery lock for target client;

     return message list to pool;

}
```

FIGURE 47 Message Delivery Algorithm.

The message delivery algorithm can also be called directly from a CIT if the target client is the CIT's own client, thus, a message delivery lock may actually be synchronizing a CIT and any COTs that need to deliver messages to the same client.

Note that a counter is maintained for each target client indicating the number of flush attempts that have occured since the last successful flush. This value could be used to determine if a client is not reading its messages and an attempted flush to this client has blocked on the writev.

12.5.5 Grab Message Delivery Implementation

The Grab Message Delivery is a limited mechanism. It is intended for use with large replies that cannot, or do not want to, send all the data to the target client in one chunk of memory. GetImage is an example of a request with a potentially large reply that will use this mechnism, see Section 12.5.5.1. Since a reply will only be written by a CIT to its own client, the grab implementation only allows a CIT to grab message delivery for its client.Also, grabbing message delivery should only occur after all other messages to be generated by that request have been locally buffered.

Grabbing message delivery will allow the owner of that grab exclusive access to the output data flow to the grabbed client. No other messages will be delivered to that client until the grab is released.

Initiating a message delivery grab will cause all messages in the global buffer to be flushed. Following that, the message delivery mutex for the target client is grabbed, assuring that no other thread can now write to the target client.

At this point, the design has to be careful not to violate the X protocol requirements for output flow (see Section 12.3.2). The grabbing client may have some messages buffered in the local buffer and still has a reply to write to complete the operation. The global buffer may have new messages that arrived during the window between the flush just performed by the grab and this point. Any such messages for the target client have to be delivered before the locally buffered messages.

The global message buffer is locked followed by a transfer of all locally buffered messages to the global buffer. Then, while still holding the lock, the list of messages for the target client are removed from the global message buffer. Finally, the global message buffer lock is released. Holding the lock over both operations will assure that no messages are added to the target client's message list between the transfer and the removing of the target list from the global buffer. If new messages were added to the target client's list during that window, the resulting flow of output would contain a mix of messages

from two different operations, violating the first X protocol condition. The entire algorithm is shown in FIGURE 48.

```
GrabClientMessageDelivery
{
        verify valid CIT request;
        flush all pending messages in global buffer;
        obtain message delivery lock for target client;
        obtain global message buffer lock;
        transfer all locally buffered messages to global buffer;
        get message list for target client from global buffer;
        release global message buffer lock;
        write messages to target client;
}


UngrabClientMessageDelivery
{
        verify valid CIT request;
        release message delivery lock for target client;
}
```

FIGURE 48 Grab and Ungrab Message Delivery Algorithms.

12.5.5.1 ProcGetImage Implementation

GetImage is the only core request that utilizes the grab and ungrab message delivery. If the length of the reply data for this request exceeds a predefined amount, IMAGE_BUFSIZE, or if the memory allocation for the reply data fails, the request grabs client message delivery and then allocates a buffer locally on the stack. It will first attempt to allocate a buffer of size IMAGE_BUFSIZE, and retry on failure with a buffer half the size of the previous attempt. It will retry until it allocates a buffer or reaches a minimum buffer size. If the buffer cannot be allocated, BadAlloc is returned. If the buffer is allocated, the reply data is sent directly to the client in chunks. When all the data has been written, the grab is released.

If the total reply data size does not exceed the IMAGE_BUFSIZE and enough memory for the reply data can be allocated, it will be handled via the normal message delivery mechanism and no grab is acquired.

CHAPTER 13 Other MTX Locking

13.1 Font Subsystem

The X-server contains a font subsystem that supplies direct support for all font operations or communicates with a Font Server to carry out font operations. This subsystem contains several data objects that require locking that is not provided by the previous locking mechanisms presented in this document. In this section, we will discuss the structure of the font subsystem, the locking requirements, the current locking design, and alternative locking methods. FIGURE 49 illustrates the various components involved in handling fonts.

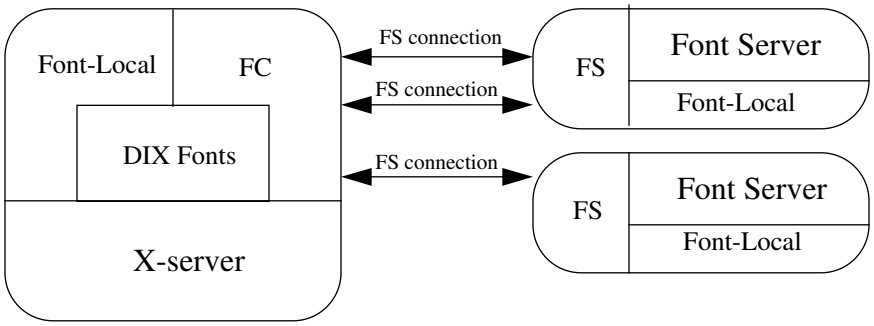


FIGURE 49 Components in Font Subsystem

FIGURE 49 contains the following components:

- **DIX Fonts:** This component is contained only in the X-server. It contains top-level routines and data objects for font support. Some operations require lower-level support that may be provided locally or by a font server. At this high level, the code and data objects have no dependency on knowledge of whether or not a font server will be involved in completing an operation. When an operation proceeds to the point where lower-level support is needed, the DIX Fonts code dispatches to either the font server interface routines (FC) or to routines that handle the operation locally (Font-Local). The DIX Fonts routines are not reentrant, locking is required.
- **Font-Local:** This code implements the actual font operations such as OpenFont, CloseFont, ListFont, and ListFontWithInfo. This code is common between the X-server and a Font Server. Note that if a font operation is handled locally by the X-server, the Font-Local component will be entered. This code is not reentrant, locking is required on the X-server side.
- **FC layer:** This component is contained only in the X-server. It contains the interface routines to a font server. This code is will dispatch requests to a font server to carry out such operations as OpenFont, CloseFont, ListFont, and ListFontWithInfo. This code is not reentrant, locking is required.
- **FS layer:** Interface to the X-server. Reads requests from the X-server and writes data back to it. Code and data objects in this component are not an issue in the design of the MTX server.

13.1.1 Locking Alternatives

Method 1: One locking option would be to allow a request that requires entry into the font subsystem exclusive access to the subsystem. An exclusive POQ lock is a simple implementation of this option. This is a very coarse grained locking scheme. Since the subsystem is expected to be entered infrequently it may seem to be a reasonable option. One problem with this method is that the exclusive lock would be held while a request is issued to a font server, which could take an arbitrarily long time to execute depending on network delays or failures.

Method 2: Another locking option would be to allow concurrent access to the DIX Font component of the font subsystem by adding appropriate locks to make it reentrant and then serializing all requests to the lower level components with an exclusive lock at the point of dispatch. This method provides a little more concurrency than Method 1, but suffers from the same problem described in Method 1 (holding an exclusive lock during a font server request).

Method 3: Another locking option would be to allow concurrent access to the DIX Font component and the FC component of the font subsystem by adding appropriate locks to make them reentrant. With this method, requests to the same font server connection would be serialized by an exclusive lock associated with that individual connection. Likewise, requests to be handled locally in the Font-Local component would be serialized. This method would provide a reasonable amount of concurrency without the problems described in Methods 1 and 2.

NOTE: A method with finer grained locking than all the previous methods may be possible, but may not be worth the added complexities. This document will not attempt to describe other possibilities.

13.1.2 Current Locking Mechanism

While method 3, as discussed in Section 13.1.1, is considered the preferred locking method, it was not considered until late in the development cycle and is not implemented in the alpha snapshot. Method 2 is currently implemented.

13.1.3 Implementation details

13.1.3.1 Method 2

With Method 2, the DIX Fonts component must be made reentrant. This component has two global data objects that must be protected, the FontPathElement linked list and the FontPatternCache. The FontPatternCache is protected by obtaining a single exclusive lock (FontMutex) during access. The FontPathElement linked list is protected by Read/Write POQ locks (see CHAPTER 10).

Method 2 also requires that requests to the lower-level Font-Local and FC components be serialized. This is achieved by obtaining an exclusive lock before dispatching to the lower level components and releasing it upon return (FPEFuncMutex).

13.1.3.2 Method 3

With Method 3, the DIX Fonts and FC components must be made reentrant. Also exclusive locks have to be maintained for each font server connection and another lock for the local handling of fonts (the Font-Local component). This implies the Font-Local component is protected because all access to it is serialized. The DIX Fonts component will be made reentrant by the same mechanism used to implement Method 2 (see Section 13.1.3.1).

The DIX Fonts component dispatches to one of the lower-level components via a function pointer contained in the fpe_functions array. The array is indexed by a value associated with each individual lower-level component. There are function pointers for all the

font operations that are handled at that level, such as OpenFont, CloseFont, ListFont, and ListFontWithInfo. With each dispatch, a FontPathElement pointer is passed down to the lower-level. If dispatching into the FC component, the FontPathElement will contain information about a specific font server connection. An exclusive mutex could be associated with the connection information of each of these FontPathElements. This mutex would be used to serialize all requests to a specific connection. The DIX Fonts component dispatches to the lower-level components by issuing a function call of the following form.

*(*fpe_functions[fpe_type].open_font) (<parameters>, fpe, <parameters>);*

New routines would be created for each lower-level component and pointers to them stored in the fpe_functions structures in the new fields *obtain_lock* and *release_lock*. Locking for the Font-Local component would be provided by dispatching to a routine that aquires its associated exclusive lock. Locking for font server connections would be provided by dispatching to a routine that obtains the exclusive lock associated with the font server connection.

*(*fpe_functions[fpe_type].obtain_lock) (fpe);*

The last remaining problem to be solved with this method would be making the FC component reentrant. From initial investigations it appears this may just be a matter of protecting a single global variable, *awaiting_reconnect*. At the time of this writing, the investigation had not proceeded far enough to determine if awaiting_reconnect could be made obsolete in the MTX implementation, other global variables in this component became obsolete by the removal of block and wakeup handlers.

One last concern. Are the FontPathElements that are passed to the lower-level routines being properly protected? They are also referenced in the DIX Fonts component. This had not been verified, as of this writing.

13.2 Atom Database

An atom is a 32bit unique identifier that corresponds to a named property. Atoms are used so that long strings corresponding to the property names and property types is not continuously passed across the client/server connection. The atom, then, is a short cut way of expressing properties that is similar to the resource id of server objects.

All core atoms are predefined by the Main Server Thread in *InitAtoms*. *InitAtoms* calls *MakePredeclaredAtoms* to build the initial core table. The Atom Database is global to the server. The Atom Database is composed of the Atom Table and a binary tree of atom nodes. The table is an array of pointers to atom records indexed by the atom numbers as defined in Xatom.h. Core atom numbers start at 1 and end at 68. User defined atoms start at 69. The initial table size is 100. XA_LAST_PREDEFINED always points to the last entry in the table.

Atoms can be added to the Atom Database one at a time, but the database is never modified except when it is reinitialized. This happens when the Main Server Thread resets the server.

Each array element in the AtomTable points to a node in the Atom Binary Tree. The Atom Binary Tree exists to expedite searches of the AtomTable given a property string. This tree is composed of nodes that contain pointers to the next left and next right nodes, the atom number, the fingerprint, and the property string. This fingerprint is a number based on the contents of the property name/type string. Through a magic formula, the string is converted to a fingerprint that is used as the key when traversing the tree. Low fingerprints to the left; high fingerprints to the right.

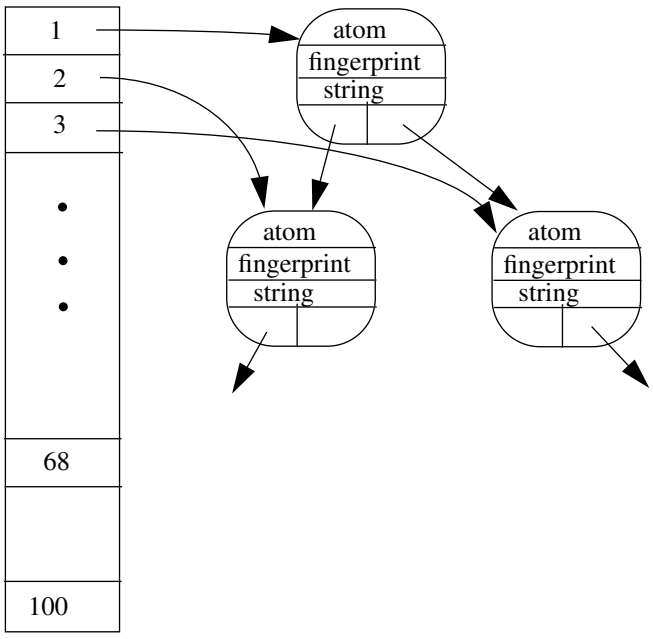


FIGURE 50 Atom Database

The *InternAtom* protocol request looks up the atom number for a given property. If the atom doesn't exist, the request specifies through *onlyIfExists* if it should be added to the Atom Table if not found. This protocol request may need to write to the table if the X Client is adding a user property. NOTE: The ICCCM documentation suggests that an X Client never go to the server when querying about core atoms. The X Client should know the atoms and properties on its side.

The GetAtom protocol request looks up the property string based on a given atom number. The AtomTable only needs to be read.

The AtomTable is accessed infrequently and for short durations. Therefore, an exclusive lock on the AtomTable would be sufficient. If testing proves otherwise, performance could be increased by using a reader/writer lock. The course grained exclusive lock is implemented for the atom table on the POQ (see CHAPTER 10).