InterSystems®
Caché

Locking and Concurrency Control

Version 2018.1
2018-12-10

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Locking and Concurrency Control
Caché Version 2018.1 2018-12-10
Copyright © 2018 InterSystems Corporation
All rights reserved.

Table of Contents

Locking and Concurrency Control

An important feature of any multi-process system is concurrency control, the ability to prevent different processes from changing a specific element of data at the same time, resulting in corruption. Consequently, ObjectScript provides a lock management system. This article provides an overview. It discusses the following topics:

- Overview of locking
- Locks and arrays
- How to use the LOCK command
- Types of locks
- Additional information about escalating locks
- Locks, globals, and namespaces
- Avoiding deadlock
- Practical uses for locks
- Additional sources of information

Caché SQL, Caché MVBasic, and Caché Basic also provide commands for working with locks. For details, see the *Caché SQL Reference*, the *Caché MultiValue Basic Reference*, and the *Caché Basic Reference*.

Also, the %Persistent class provides a way to control concurrent access to objects, namely, the concurrency argument to **%OpenId()** and other methods of this class. These methods ultimately use the ObjectScript **LOCK** command, which is discussed in this article. All persistent objects inherit these methods. See "Object Concurrency" in *Using Caché Objects*. Similarly, the system automatically performs locking on INSERT, UPDATE, and DELETE operations (unless you specify the %NOLOCK keyword).

The %Persistent class also provides the methods **%GetLock()**, **%ReleaseLock()**, **%LockId()**, **%UnlockId()**, **%LockExtent()**, and **%UnlockExtent()**. For details, see the class reference for %Persistent.

1 Introduction

The basic locking mechanism is the **LOCK** command. The purpose of this command is to delay activity in one process until another process has signaled that it is OK to proceed.

In Caché, a lock does not, by itself, prevent activity. Locking works only by convention: it requires that mutually competing processes all implement locking with the same lock names. For example, the following describes a common scenario:

1. Process A issues the **LOCK** command, and Caché creates a lock (by default, an exclusive lock).

   Typically, process A then makes changes to nodes in a global. The details are application-specific.

2. Process B issues the **LOCK** command with the same lock name. Because there is an existing exclusive lock, process B pauses. Specifically, the **LOCK** command does not return, and no successive lines of code can be executed.
3. When the process A releases the lock, the **LOCK** command in process B finally returns and process B continues.

   Typically, process B then makes changes to nodes in the same global.

1.1 Lock Names

One of the arguments for the **LOCK** command is the lock name. Lock names are arbitrary, but by universal convention, programmers use lock names that are identical to the names of the item to be locked. Usually the item to be locked is a global or a node of a global. Thus lock names usually look like names of global names or names of nodes of globals. (This article discusses only lock names that start with carets, because those are the most common; for details on locks with name that do not start with carets, see “LOCK” in the *Caché ObjectScript Reference*.)

Formally, lock names follow the same naming conventions as local variables and global variables, as described in the chapter “Variables” in *Using Caché ObjectScript*. Like variables, lock names are case-sensitive and can have subscripts. Do not use process-private global names as lock names (you would not need such a lock anyway because by definition only one process can access such a global).

Tip: Because locking works by convention and because lock names are arbitrary, it is not necessary to define a given variable *before* creating a lock with the same name.

The form of the lock name has an effect on performance, because of how Caché allocates and manages memory. Locking is optimized for lock names that use subscripts. An example is `^sample.person(id)`.

In contrast, Caché is not optimized for lock names such as `^name_concatenated_identifier`. Non-subscripted lock names can also cause performance problems related to ECP.

1.2 The Lock Table

Caché maintains a system-wide, in-memory table that records all current locks and the processes that have own them. This table — the lock table — is accessible via the Management Portal, where you can view the locks and (in rare cases, if needed) remove them. Note that any given process can own multiple locks, with different lock names (or even multiple locks with the same lock name).

When a process ends, the system automatically releases all locks that the process owns. Thus it is not generally necessary to remove locks via the Management Portal, except in the case of an application error.

The lock table cannot exceed a fixed size, which you can specify. For information, see “Monitoring Locks” in the *Caché Monitoring Guide*. Consequently, it is possible for the lock table to fill up, such that no further locks are possible. If this occurs, Caché writes the following message to the cconsole.log file:

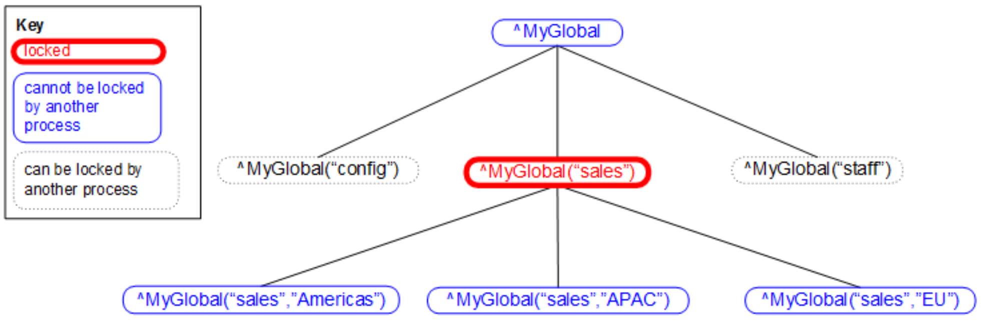
```
LOCK TABLE FULL
```

Filling the lock table is *not* generally considered to be an application error; Caché also provides a lock queue, and processes wait until there is space to add their locks to the lock table. (However, deadlock *is* considered an application programming error. See “Avoiding Deadlock,” later in this article.)

2 Locks and Arrays

When you lock an array, you can lock either the entire array or one or more nodes in the array. When you lock an array node, other processes are blocked from locking any node that is subordinate to that node. Other processes are also blocked from locking the direct ancestors of the locked node.

The following figure shows an example:


Key
locked
cannot be locked by another process
can be locked by another process
^MyGlobal
^MyGlobal("config")
^MyGlobal("sales")
^MyGlobal("staff")
^MyGlobal("sales","Americas")
^MyGlobal("sales","APAC")
^MyGlobal("sales","EU")

Implicit locks are not included in the lock table and thus do not affect the size of the lock table.

The Caché lock queuing algorithm queues all locks for the same lock name in the order received, even when there is no direct resource contention. For an example and details, see "Queuing of Array Node Locks" in the chapter "Lock Management" of *Using Caché ObjectScript*.

3 Using the LOCK Command

This section discusses how to use the LOCK command to add and remove locks.

3.1 Adding an Incremental Lock

To add a lock, use the **LOCK** command as follows:

```
LOCK +lockname
```

Where *lockname* is the literal lock name. The plus sign (+) creates an incremental lock, which is the common scenario; see "Creating Simple Locks" for a less common alternative.

This command does the following:

1. Attempts to add the given lock to the lock table. That is, this entry is added to the lock queue.
2. Pauses execution until the lock can be acquired.

There are different types of locks, which behave differently. To add a lock of a non-default lock type, use the following variation:

```
LOCK +lockname#locktype
```

Where *locktype* is a string of lock type codes enclosed in double quotes; see the later section "Lock Types."

Note that a given process can add multiple incremental locks with the same name; these locks can be of different types or can all be the same type.

3.2 Adding an Incremental Lock with a Timeout

If used incorrectly, incremental locks can result in an undesirable situation known as *deadlock*, discussed later in "Avoiding Deadlock." One way to avoid deadlock is to specify a timeout period when you create a lock. To do so, use the **LOCK** command as follows:

```
LOCK +lockname#locktype :timeout
```

Where *timeout* is the timeout period in seconds. The space before the colon is optional. If you specify *timeout* as 0, Caché makes one attempt to add the lock (but see the note, below).

This command does the following:

1. Attempts to add the given lock to the lock table. That is, this entry is added to the lock queue.
2. Pauses execution until the lock can be acquired or until the timeout period ends, whichever comes first.
3. Sets the value of the **$TEST** special variable. If the lock is acquired, Caché sets **$TEST** equal to 1. Otherwise, Caché sets **$TEST** equal to 0.

This means that if you use the timeout argument, your code should next check the value of the **$TEST** special variable and use the value to choose whether to proceed. The following shows an example:

```
 Lock +^ROUTINE(routinename):0
 If '$TEST {  Return $$$ERROR("Cannot lock the routine: ",routinename)}
```

3.2.1 A Note on the Zero Timeout

As noted above, if you specify *timeout* as 0, Caché makes one attempt to add the lock. However, if you try to take a lock on a parent node using a zero timeout, and you already have a lock on a child node, the zero timeout is ignored and there is an internal 1 second timeout, which is used instead.

3.3 Removing a Lock

To remove a lock of the default type, use the **LOCK** command as follows:

```
LOCK -lockname
```

If the process that executes this command owns a lock (of the default type) with the given name, this command removes that lock. Or if the process owns more than one lock (of the default type), this command removes one of them.

Or to remove a lock of another type:

```
LOCK -lockname#locktype
```

Where *locktype* is a string of lock type codes; see the later section "Lock Types." The lock type codes do *not* have to be in the same order as when the lock was created.

3.4 Other Basic Variations of the LOCK Command

For completeness, this section discusses the other basic variations of the **LOCK** command: using it to create simple locks and using it to remove all locks. These variations are uncommon in practice.

3.4.1 Creating Simple Locks

For the **LOCK** command, if you omit the **+** operator, the **LOCK** command first removes all existing locks held by this process and then attempts to add the new lock. In this case, the lock is called a *simple lock* rather than an incremental lock. It is possible for a process to own multiple simple locks, *if* that process creates them all at the same time with syntax like the following:

```
 LOCK (^MyVar1,^MyVar2,^MyVar3)
```

Simple locks are not common in practice, because it is usually necessary to hold multiple locks and to acquire them at different steps in your code. Thus it is more practical to use incremental locks.

However, if simple locks are appropriate for you, note that you can specify the *locktype* and *timeout* arguments when you create a simple lock. Also, to remove a simple lock, you can use the **LOCK** command with a minus sign (`-`).

3.4.2 Removing All Locks

To remove all locks held by the current process, use the **LOCK** command with no arguments. In practice, it is not common to use the command this way, for two reasons:

- It is best to release specific locks as soon as possible.
- When the process ends, all its locks are automatically released.

4 Lock Types

The *locktype* argument specifies the type of lock to add or remove. When adding a lock, include this argument as follows:

```
LOCK +lockname#locktype
```

Or when removing a lock:

```
LOCK -lockname#locktype
```

In either case, *locktype* is one or more lock type codes (in any order) enclosed in double quotes. Note that if you specify the *locktype* argument, you must include a pound character (#) to separate the lock name from the lock type.

There are four lock type codes, as follows. Note that these are not case-sensitive.

- `S` — Adds a shared lock. See "Exclusive and Shared Locks."
- `E` — Adds an escalating lock. See "Non-Escalating and Escalating Locks."
- `I` — Adds a lock with immediate unlock.
- `D` — Adds a lock with deferred unlock.

  The lock type codes `D` and `I` have special behavior in transactions. For details, see LOCK in the *Caché ObjectScript Reference*. You cannot use these two lock type codes at the same time for the same lock name.

The next sections discuss the most common variations, and the last subsection summarizes all the lock types.

4.1 Exclusive and Shared Locks

Any lock is either *exclusive* (the default) or *shared*. These types have the following significance:

- While one process owns an exclusive lock (with a given lock name), no other process can acquire any lock with that lock name.
- While one process owns a shared lock (with a given lock name), other processes can acquire shared locks with that lock name, but no other process can acquire an exclusive lock with that lock name.

The typical purpose of an exclusive lock is to indicate that you intend to modify a value and that other processes should not attempt to read or modify that value. The typical purpose of a shared lock is to indicate that you intend to read a value and that other processes should not attempt to modify that value; they can, however, read the value. Also see the later section "Practical Uses for Locks."

4.2 Non-Escalating and Escalating Locks

Any lock is also either *non-escalating* (the default) or *escalating*. The purpose of escalating locks is to make it easier to manage large numbers of locks, which consume memory and which increase the chance of filling the lock table.

You use escalating locks when you lock multiple nodes of the same array. For escalating locks, if a given process has created more than a specific number (by default, 1000) of locks on parallel nodes of a given array, Caché replaces the individual lock names and replaces them with a new lock that contains the lock count. (In contrast, Caché never does this for non-escalating locks.) For an example and additional details, see the later section "Escalating Locks."

Note: You can create escalating locks only for lock names that include subscripts. If you attempt to create an escalating lock with a lock name that has no subscript, Caché issues a <COMMAND> error.

4.3 Summary of Lock Types

The following table lists all the possible lock types with their descriptions:

	Exclusive Locks	Shared Locks (`#"S"` locks)
Non-escalating Locks	• *locktype omitted* — Default lock type • `#"I"` — Exclusive lock with immediate unlock • `#"D"` — Exclusive lock with deferred unlock	• `#"S"` — Shared lock • `#"SI"` — Shared lock with immediate unlock • `#"SD"` — Shared lock with deferred unlock
Escalating Locks (`#"E"` locks)	• `#"E"` — Exclusive escalating lock • `#"EI"` — Exclusive escalating lock with immediate unlock • `#"ED"` — Exclusive escalating lock with deferred unlock	• `#"SE"` — Shared escalating lock • `#"SEI"` — Shared escalating lock with immediate unlock • `#"SED"` — Shared escalating lock with deferred unlock

For any lock type that uses multiple lock codes, the lock codes can be in any order. For example, the lock type `#"SI"` is equivalent to `#"IS"`.

For details on immediate unlock and deferred unlock, see LOCK in the *Caché ObjectScript Reference*. You cannot use these two lock type codes at the same time for the same lock name.

5 Escalating Locks

You use escalating locks to manage large numbers of locks. They are relevant when you lock nodes of an array, specifically when you lock multiple nodes at the same subscript level.

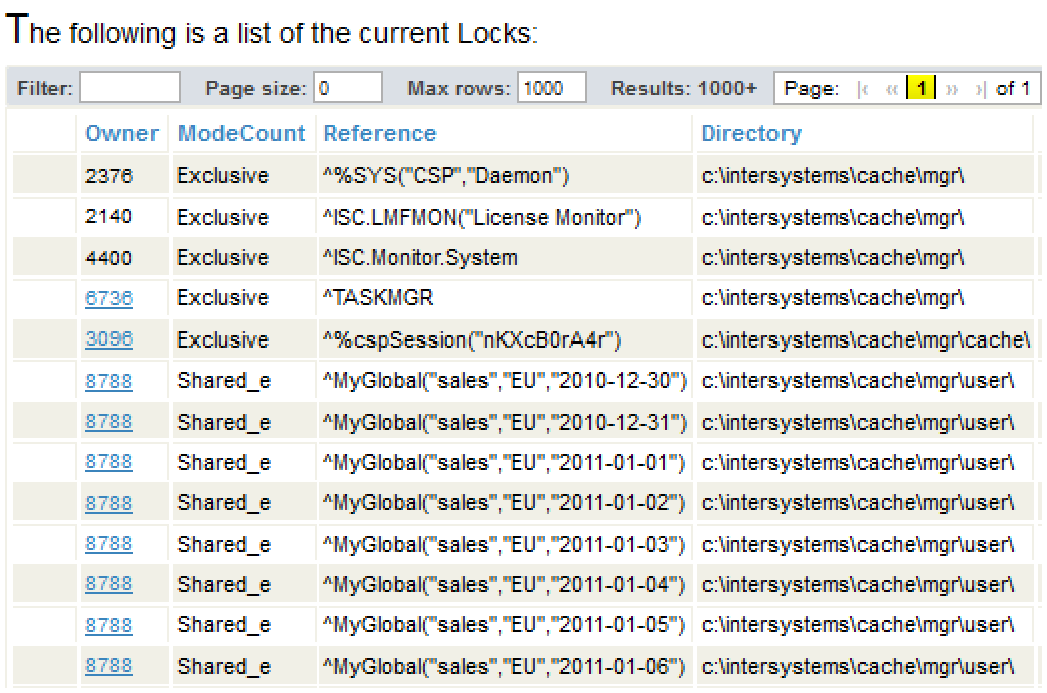
When a given process has created more than a specific number (by default, 1000) of escalating locks at a given subscript level in the same array, Caché removes all the individual lock names and replaces them with a new lock. The new lock is at the parent level, which means that this entire branch of the array is implicitly locked. The example (shown next) demonstrates this.

Your application should release locks for specific child nodes as soon as it is suitable to do so (exactly as with non-escalating locks). As you release locks, Caché decrements the corresponding lock count. When your application removes enough locks, Caché removes the lock on the parent node. The second subsection shows an example.

For information on specifying the lock threshold (which by default is 1000), see "LockThreshold" in the *Caché Parameter File Reference*.

5.1 Lock Escalation Example

Suppose that you have 1000 locks of the form `^MyGlobal("sales","EU",salesdate)` where *salesdate* represents dates. The lock table might look like this:

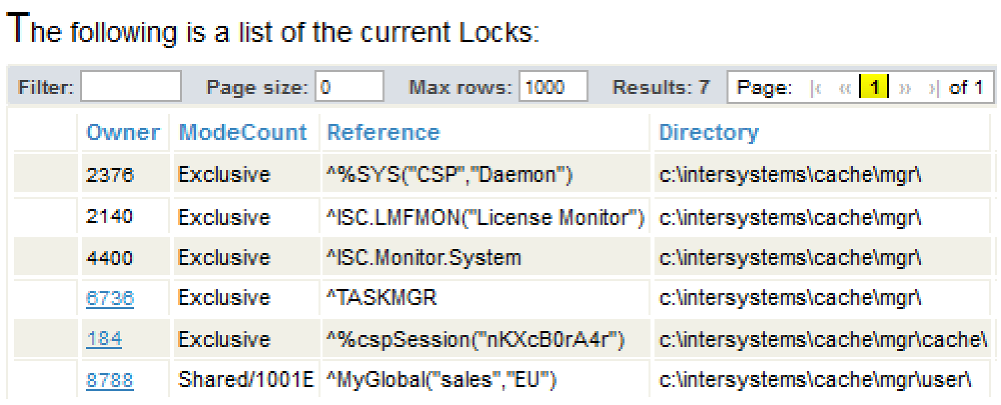

The following is a list of the current Locks:

Filter: Page size: 0 Max rows: 1000 Results: 1000+ Page: |« « 1 » »| of 1

Owner	ModeCount	Reference	Directory
2376	Exclusive	^%SYS("CSP","Daemon")	c:\intersystems\cache\mgr\
2140	Exclusive	^ISC.LMFMON("License Monitor")	c:\intersystems\cache\mgr\
4400	Exclusive	^ISC.Monitor.System	c:\intersystems\cache\mgr\
6736	Exclusive	^TASKMGR	c:\intersystems\cache\mgr\
3096	Exclusive	^%cspSession("nKXcB0rA4r")	c:\intersystems\cache\mgr\cache\
8788	Shared_e	^MyGlobal("sales","EU","2010-12-30")	c:\intersystems\cache\mgr\user\
8788	Shared_e	^MyGlobal("sales","EU","2010-12-31")	c:\intersystems\cache\mgr\user\
8788	Shared_e	^MyGlobal("sales","EU","2011-01-01")	c:\intersystems\cache\mgr\user\
8788	Shared_e	^MyGlobal("sales","EU","2011-01-02")	c:\intersystems\cache\mgr\user\
8788	Shared_e	^MyGlobal("sales","EU","2011-01-03")	c:\intersystems\cache\mgr\user\
8788	Shared_e	^MyGlobal("sales","EU","2011-01-04")	c:\intersystems\cache\mgr\user\
8788	Shared_e	^MyGlobal("sales","EU","2011-01-05")	c:\intersystems\cache\mgr\user\
8788	Shared_e	^MyGlobal("sales","EU","2011-01-06")	c:\intersystems\cache\mgr\user\

Notice the entries for process 8788. The **ModeCount** column indicates that these are shared, escalating locks.

When the same process attempts to create another lock of the same form, Caché escalates them. It removes these locks and replaces them with a single lock of the name `^MyGlobal("sales","EU")`. Now the lock table might look like this:


The following is a list of the current Locks:

Filter: Page size: 0 Max rows: 1000 Results: 7 Page: |« « 1 » »| of 1

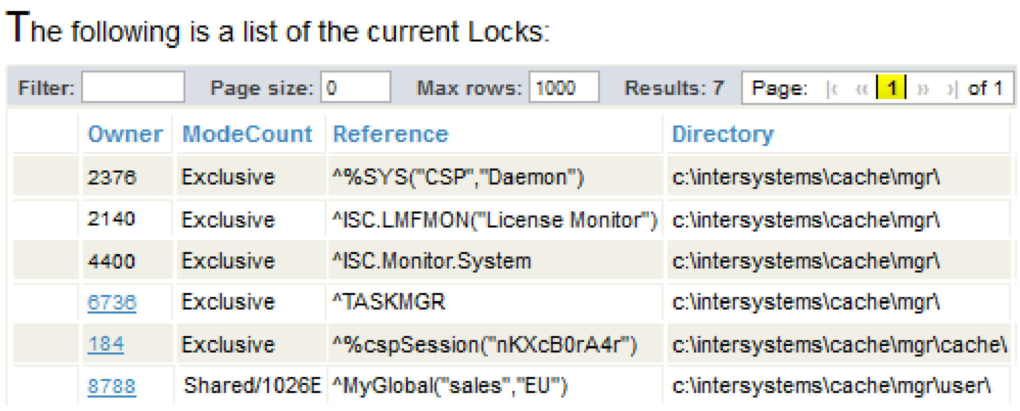
Owner	ModeCount	Reference	Directory
2376	Exclusive	^%SYS("CSP","Daemon")	c:\intersystems\cache\mgr\
2140	Exclusive	^ISC.LMFMON("License Monitor")	c:\intersystems\cache\mgr\
4400	Exclusive	^ISC.Monitor.System	c:\intersystems\cache\mgr\
6736	Exclusive	^TASKMGR	c:\intersystems\cache\mgr\
184	Exclusive	^%cspSession("nKXcB0rA4r")	c:\intersystems\cache\mgr\cache\
8788	Shared/1001E	^MyGlobal("sales","EU")	c:\intersystems\cache\mgr\user\

The **ModeCount** column indicates that this is a shared, escalating lock and that its count is 1001.

Note the following key points:

- All child nodes of `^MyGlobal("sales","EU")` are now implicitly locked, following the basic rules for array locking.
- The lock table no longer contains information about which child nodes of `^MyGlobal("sales","EU")` were specifically locked. This has important implications when you remove locks; see the next subsection.

When the same process adds more lock names of the form `^MyGlobal("sales","EU",salesdate)`, the lock table increments the lock count for the lock name `^MyGlobal("sales","EU")`. The lock table might then look like this:



The following is a list of the current Locks:

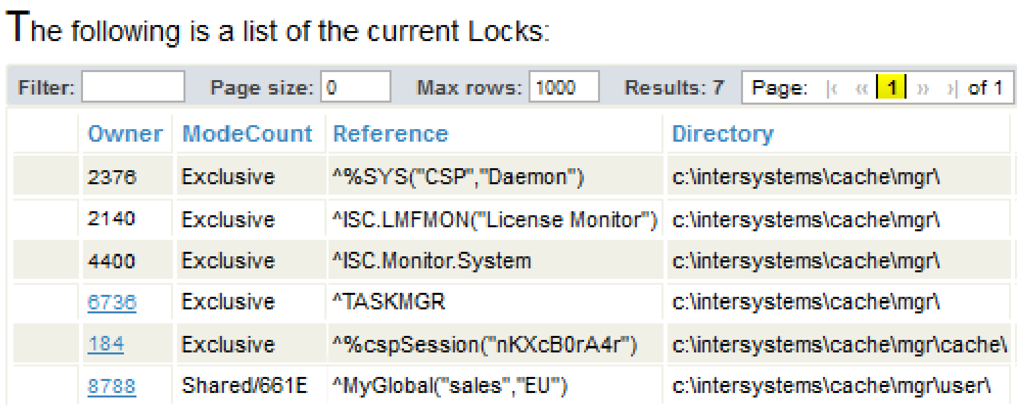
Filter: Page size: 0 Max rows: 1000 Results: 7 Page: 1 of 1

	Owner	ModeCount	Reference	Directory
	2376	Exclusive	^%SYS("CSP","Daemon")	c:\intersystems\cache\mgr\
	2140	Exclusive	^ISC.LMFMON("License Monitor")	c:\intersystems\cache\mgr\
	4400	Exclusive	^ISC.Monitor.System	c:\intersystems\cache\mgr\
	6736	Exclusive	^TASKMGR	c:\intersystems\cache\mgr\
	184	Exclusive	^%cspSession("nKXcB0rA4r")	c:\intersystems\cache\mgr\cache\
	8788	Shared/1026E	^MyGlobal("sales","EU")	c:\intersystems\cache\mgr\user\

The **ModeCount** column indicates that the lock count for this lock is now 1026.

5.2 Removing Escalating Locks

In exactly the same way as with non-escalating locks, your application should release locks for specific child nodes as soon as possible. As you do so, Caché decrements the lock count for the escalated lock. For example, suppose that your code removes the locks for `^MyGlobal("sales","EU",salesdate)` where *salesdate* corresponds to any date in 2011 — thus removing 365 locks. The lock table now looks like this:



The following is a list of the current Locks:

Filter: Page size: 0 Max rows: 1000 Results: 7 Page: 1 of 1

	Owner	ModeCount	Reference	Directory
	2376	Exclusive	^%SYS("CSP","Daemon")	c:\intersystems\cache\mgr\
	2140	Exclusive	^ISC.LMFMON("License Monitor")	c:\intersystems\cache\mgr\
	4400	Exclusive	^ISC.Monitor.System	c:\intersystems\cache\mgr\
	6736	Exclusive	^TASKMGR	c:\intersystems\cache\mgr\
	184	Exclusive	^%cspSession("nKXcB0rA4r")	c:\intersystems\cache\mgr\cache\
	8788	Shared/661E	^MyGlobal("sales","EU")	c:\intersystems\cache\mgr\user\

Notice that even though the number of locks is now below the threshold (1000), the lock table does not contain individual entries for the locks for `^MyGlobal("sales","EU",salesdate)`.

The node `^MyGlobal("sales")` remains explicitly locked until the process removes 661 more locks of the form `^MyGlobal("sales","EU",salesdate)`.

Important: There is a subtle point to consider, related to the preceding discussion. It is possible for an application to “release” locks on array nodes that were never locked in the first place, thus resulting in an inaccurate lock count for the escalated lock — and possibly releasing the escalated lock before it is desirable to do so.

For example, suppose that the process locked nodes in `^MyGlobal("sales","EU",salesdate)` for the years 2010 through the present. This would create more than 1000 locks and this lock would be escalated, as planned. Suppose that a bug in the application removes locks for the nodes for the year 1970. Caché would permit this action, even though those nodes were not previously locked, and Caché would decrement the lock count by 365. The resulting lock count would not be an accurate count of the desired locks. If the application then removed locks for other years, the escalated lock could potentially be removed unexpectedly early.

6 Locks, Globals, and Namespaces

Locks are typically used to control access to globals. Because a global can be accessed from multiple namespaces, Caché provides automatic cross-namespace support for its locking mechanism. The behavior is automatic and needs no intervention, but is described here for reference. There are several scenarios to consider:

- Any namespace has a default database which contains data for persistent classes and any additional globals; this is the *global database* for this namespace. When you access data (in any manner), Caché retrieves it from this database unless other considerations apply. A given database can be the global database for more than one namespace. See Scenario 1.
- A namespace can include mappings that provide access to globals stored in other databases. See Scenario 2.
- A namespace can include subscript level global mappings that provide access to globals partly stored in other databases. See Scenario 3.
- Code running in one namespace can use an extended reference to access a global not otherwise available in this namespace. See Scenario 4.

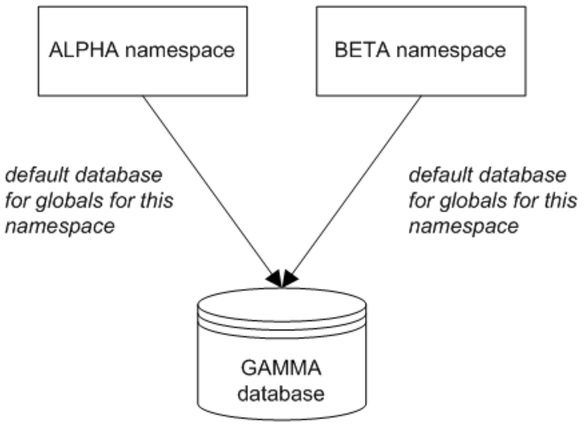
Although lock names are intrinsically arbitrary, when you use a lock name that starts with a caret (`^`), Caché provides special behavior appropriate for these scenarios. The following subsections give the details. For simplicity, only exclusive locks are discussed; the logic is similar for shared locks.

6.1 Scenario 1: Multiple Namespaces with the Same Global Database

As noted earlier, while process A owns an exclusive lock with a given lock name, no other process can acquire any lock with the same lock name.

If the lock name starts with a caret, this rule applies to *all* namespaces that use the same global database.

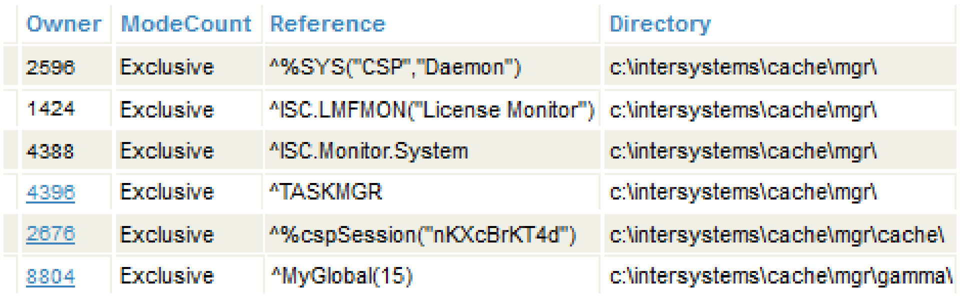
For example, suppose that the namespaces `ALPHA` and `BETA` are both configured to use database `GAMMA` as their global database. The following shows a sketch:


ALPHA namespace
BETA namespace
default database for globals for this namespace
default database for globals for this namespace
GAMMA database

Then consider the following scenario:

1. In namespace ALPHA, process A acquires an exclusive lock with the name `^MyGlobal(15)`.
2. In namespace BETA, process B tries to acquire a lock with the name `^MyGlobal(15)`. This **LOCK** command does not return; the process is blocked until process A releases the lock.

In this scenario, the lock table contains only the entry for the lock owned by Process A. If you examine the lock table, you will notice that it indicates the *database* to which this lock applies; see the **Directory** column. For example:

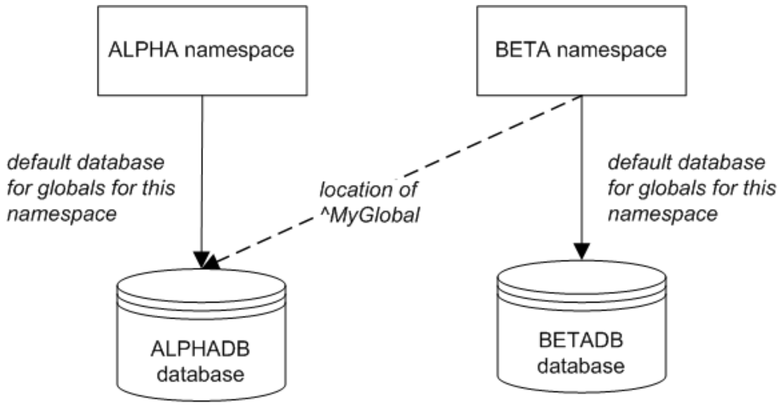


Owner	ModeCount	Reference	Directory
2596	Exclusive	^%SYS("CSP","Daemon")	c:\intersystems\cache\mgr\
1424	Exclusive	^ISC.LMFMON("License Monitor")	c:\intersystems\cache\mgr\
4388	Exclusive	^ISC.Monitor.System	c:\intersystems\cache\mgr\
4396	Exclusive	^TASKMGR	c:\intersystems\cache\mgr\
2676	Exclusive	^%cspSession("nKXcBrKT4d")	c:\intersystems\cache\mgr\cache\
8804	Exclusive	^MyGlobal(15)	c:\intersystems\cache\mgr\gamma\

6.2 Scenario 2: Namespace Uses a Mapped Global

If one or more namespaces include global mappings, the system automatically enforces the lock mechanism across the applicable namespaces. Caché automatically creates additional lock table entries when locks are acquired in the non-default namespace.

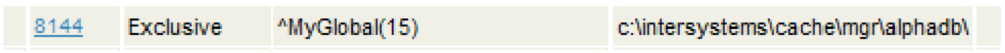
For example, suppose that namespace ALPHA is configured to use database ALPHADB as its global database. Suppose that namespace BETA is configured to use a different database (BETADB) as its global database. The namespace BETA also includes a global mapping that specifies that `^MyGlobal` is stored in the ALPHADB database. The following shows a sketch:


ALPHA namespace
BETA namespace
default database for globals for this namespace
location of ^MyGlobal
default database for globals for this namespace
ALPHADB database
BETADB database

Then consider the following scenario:

1. In namespace ALPHA, process A acquires an exclusive lock with the name `^MyGlobal(15)`.

   As with the previous scenario, the lock table contains only the entry for the lock owned by Process A. This lock applies to the ALPHADB database:

8144	Exclusive	^MyGlobal(15)	c:\intersystems\cache\mgr\alphadb\

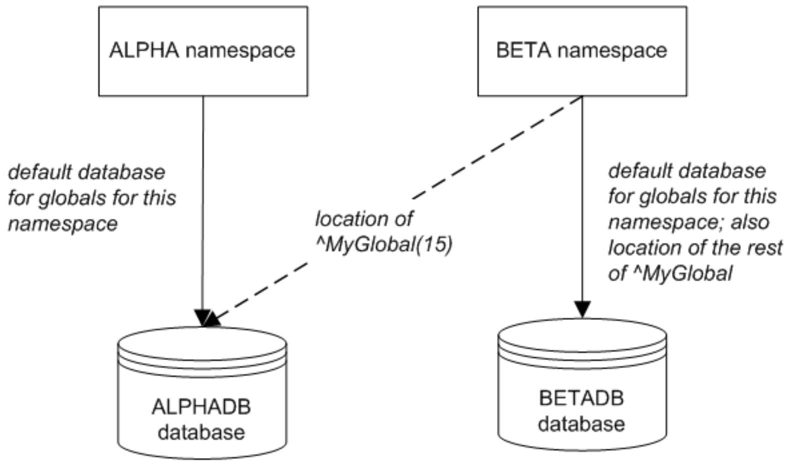
2. In namespace BETA, process B tries to acquire a lock with the name `^MyGlobal(15)`. This **LOCK** command does not return; the process is blocked until process A releases the lock.

6.3 Scenario 3: Namespace Uses a Mapped Global Subscript

If one or more namespaces include global mappings that use subscript level mappings, the system automatically enforces the lock mechanism across the applicable namespaces. In this case, Caché also automatically creates additional lock table entries when locks are acquired in a non-default namespace.

For example, suppose that namespace ALPHA is configured to use the database ALPHADB as its global database. Namespace BETA uses the BETADB database as its global database.

Also suppose that the namespace BETA also includes a subscript-level global mapping so that `^MyGlobal(15)` is stored in the ALPHADB database (while the rest of this global is stored in the namespace's default location). The following shows a sketch:


ALPHA namespace
BETA namespace
default database for globals for this namespace
location of ^MyGlobal(15)
default database for globals for this namespace; also location of the rest of ^MyGlobal
ALPHADB database
BETADB database

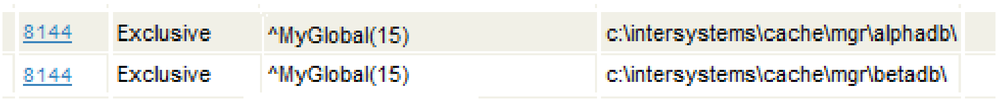
Then consider the following scenario:

1. In namespace `ALPHA`, process A acquires an exclusive lock with the name `^MyGlobal(15)`.

   As with the previous scenario, the lock table contains only the entry for the lock owned by Process A. This lock applies to the `ALPHADB` database (`C:\InterSystems\Cache\mgr\alphadb`, for example).

2. In namespace `BETA`, process B tries to acquire a lock with the name `^MyGlobal(15)`. This **LOCK** command does not return; the process is blocked until process A releases the lock.

When a non-default namespace acquires a lock, the overall behavior is the same, but Caché handles the details slightly differently. Suppose that in namespace `BETA`, a process acquires a lock with the name `^MyGlobal(15)`. In this case, the lock table contains two entries, one for the `ALPHADB` database and one for the `BETADB` database. Both locks are owned by the process in namespace `BETA`.



8144	Exclusive	^MyGlobal(15)	c:\intersystems\cache\mgr\alphadb\
8144	Exclusive	^MyGlobal(15)	c:\intersystems\cache\mgr\betadb\

When this process releases the lock name `^MyGlobal(15)`, the system automatically removes both locks.

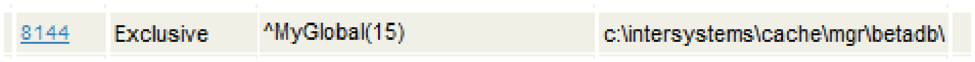
6.4 Scenario 4: Extended Global References

Code running in one namespace can use an extended reference to access a global not otherwise available in this namespace. In this case, Caché adds an entry to the lock table that affects the relevant database. The lock is owned by the process that created it. For example, consider the following scenario. For simplicity, there are no global mappings in this scenario.

1. Process A is running in the `ALPHA` namespace, and this process uses the following command to acquire a lock on a global that is available in the `BETA` namespace:

   ```
   lock ^["beta"]MyGlobal(15)
   ```

2. Now the lock table includes the following entry:

8144	Exclusive	^MyGlobal(15)	c:\intersystems\cache\mgr\betadb\

   Note that this shows only the global name (rather than the reference used to access it). Also, in this scenario, `BETADB` is the default database for the `BETA` namespace.

3. In namespace `BETA`, process B tries to acquire a lock with the name `^MyGlobal(15)`. This **LOCK** command does not return; the process is blocked until process A releases the lock.

A process-private global is technically a kind of extended reference, but Caché does not support using a process-private global names as lock names; you would not need such a lock anyway because by definition only one process can access such a global.

7 Avoiding Deadlock

Incremental locking is potentially dangerous because it can lead to a situation known as *deadlock*. This situation occurs when two processes each assert an incremental lock on a variable already locked by the other process. Because the attempted locks are incremental, the existing locks are not released. As a result, each process hangs while waiting for the other process to release the existing lock.

As an example:

1. Process A issues this command: `lock + ^MyGlobal(15)`
2. Process B issues this command: `lock + ^MyOtherGlobal(15)`
3. Process A issues this command: `lock + ^MyOtherGlobal(15)`

   This **LOCK** command does not return; the process is blocked until process B releases this lock.
4. Process B issues this command: `lock + ^MyGlobal(15)`

   This **LOCK** command does not return; the process is blocked until process A releases this lock. Process A, however, is blocked and cannot release the lock. Now these processes are both waiting for each other.

There are several ways to prevent deadlocks:

- Always include the timeout argument.
- Follow a strict protocol for the order in which you issue incremental **LOCK** commands. Deadlocks cannot occur as long as all processes follow the same order for lock names. A simple protocol is to add locks in collating sequence order.
- Use simple locking rather than incremental locking; that is, do not use the **+** operator. As noted earlier, with simple locking, the **LOCK** command first releases all previous locks held by the process. (In practice, however, simple locking is not often used.)

If a deadlock occurs, you can resolve it by using the Management Portal or the **`^LOCKTAB`** routine. See “Monitoring Locks” in the *Caché Monitoring Guide*.

8 Practical Uses for Locks

This section presents the basic ways in which locks are used in practice.

8.1 Controlling Access to Application Data

Locks are used very often to control access to application data, which is stored in globals. Your application might need to read or modify a particular piece or pieces of this data, and your application would create one or more locks before doing so, as follows:

- If your application needs to read one or more global nodes, and you do not want other processes to modify the values during the read operation, create shared locks for those nodes.
- If your application needs to modify one or more global nodes, and you do not want other processes to read these nodes during the modification, create exclusive locks for those nodes.

Then either read or make the modifications as planned. When you are done, remove the locks.

Remember that the locking mechanism works purely by convention. Any other code that would read or modify these nodes must also attempt to acquire locks before performing those operations.

8.2 Preventing Simultaneous Activity

Locks are also used to prevent multiple processes from performing the same activity. In this scenario, you also use a global, but the global contains data for the internal purposes of your application, rather than pure application data. As a simple example, suppose that you have a routine (`^NightlyBatch`) that should never be run by more than one process at any given time. This routine could do the following, at a very early stage in its processing:

1. Create an exclusive lock on a specific global node, for example, `^AppStateData("NightlyBatch")`. Specify a timeout for this operation.
2. If the lock is acquired, set nodes in a global to record that the routine has been started (as well as any other relevant information). For example:

   ```
   set ^AppStateData("NightlyBatch")=1
   set ^AppStateData("NightlyBatch","user")=$USERNAME
   ```

   Or, if the lock is not acquired within the timeout period, quit with an error message that indicates that this routine has already been started.

Then, at the end of its processing, the same routine would clear the applicable global nodes and release the lock.

The following partial example demonstrates this technique, which is adapted from code that Caché uses internally:

```
lock ^AppStateData("NightlyBatch"):0
if '$TEST {
   write "You cannot run this routine right now."
   write !, "This routine is currently being run by user: "_^AppStateData("NightlyBatch","user")
   quit
}
set ^AppStateData("NightlyBatch")=1
set ^AppStateData("NightlyBatch","user")=$USERNAME
set ^AppStateData("NightlyBatch","starttime")=$h

//main routine activity omitted from example

kill ^AppStateData("NightlyBatch")
lock -^AppStateData("NightlyBatch")
```

9 For Additional Information

For additional information on locks, see the following resources:

- "LOCK" in the *Caché ObjectScript Reference*
- "^$LOCK" in the *Caché ObjectScript Reference* (**^$LOCK** is a structured system variable that contains information about locks.)
- "Transaction Processing" in *Using Caché ObjectScript*
- "Monitoring Locks" in the *Caché Monitoring Guide*