
# Semaphores in Caché

Version 2018.1
2018-12-10

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

*Semaphores in Caché*
Caché Version 2018.1 2018-12-10






For Support questions about any InterSystems products, contact:

**InterSystems Worldwide Response Center (WRC)**
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

# Table of Contents

# Semaphores in Caché

# 1 Background

Wikipedia has this definition of semaphore: “In computer science, particularly in operating systems, a semaphore is a variable or abstract data type that is used for controlling access, by multiple processes, to a common resource in a parallel programming or a multi user environment.” A semaphore is different from a mutex (or lock). A mutex is most often used to regulate competing processes access to a single resource. A semaphore is employed when there are multiple identical copies of a resource and each of those copies can be used simultaneously by separate processes.

Consider an office supply store. It may have several copiers for its customers to use, but each copier is only used by one customer at a time. to control this, there is a set of keys that enable the machine and record usage. When a customer desires to make copies of a document, they request the key from the clerk, use the machine, and then return the key, and pay for their usage. If all the machines are in use, the customer must wait until a key is returned. The location where the keys are kept functions as the semaphore.

The example can be generalized further to include different types of copiers, perhaps distinguished by the size of the copies they can make. In this case, there will be multiple semaphores and, if the copiers have any overlap in the size of the copies they make, then a customer wishing to make copies of the common size will have two resources to draw from.

# 2 Introduction

Semaphores in Caché are shared objects used to provide fast, efficient communication between processes. Each semaphore is an instance of the class, %SYSTEM.Semaphore. A semaphore can be modeled as a shared variable holding a 64-bit non-negative integer. Operations on semaphores change the value of the variable in a synchronized way in all the processes that share it. By convention, the change in the value conveys information among the processes sharing the semaphore.

Although semaphores and locks seem to have much in common, there are advantages to using semaphores. Chief among them is the fact that semaphores cause a notification to be sent when the semaphore is granted. So a process using semaphores does not need to spend processor cycles or complicate the application logic polling the lock to check if it has been released. Furthermore, semaphores work transparently over ECP connections and are far more efficient than the exchanges needed to check locks in the same situation.

# 3 Semaphore Overview

## 3.1 Semaphore Names

Semaphores are identified by their names which are supplied as a Caché string when the semaphore is created. The names given for semaphores are expected to conform to the rules for local and global variables. A semaphore name is only to distinguish it from all other existing semaphores. Examples of valid name-strings are: `SlotsOpen`, `J(3)`, and `^pendingRequest(""j"")`.

Normally, a semaphore is stored on the instance where the semaphore is created, and is visible to all processes on that instance. However, when the semaphore name looks like the name of a global variable, the semaphore is stored on the system where the global variable (including subscripts) would be mapped. This allows such semaphores to be visible to all the processes running on the instances of an ECP system.

**Note:** The name of a semaphore is a string, but at runtime can be constructed from an expression such as `"^" _ BaseName _ "(" _ (1 + 2) _ ")"`. If BaseName contains the string, "Printers", then the semaphore name is `^Printers(3)`.

## 3.2 Semaphore Values

Semaphore value are stored as 63-bit unsigned integers and therefore a semaphore value will always be greater than or equal to zero.

The largest unsigned 63-bit integer is 9,223,372,036,854,775,807 ((2 ** 63) – 1). A semaphore cannot be incremented beyond this value, nor can it be decremented below zero.

## 3.3 Semaphore Instances and Variables

Semaphores are instances of a class derived from %SYSTEM.Semaphore. Once a semaphore is created and initialized, its OREF is usually stored in an Objectscript variable so it can be used in other operations, passed as an argument, and ultimately deleted. Although the name of the variable containing the reference to the semaphore need not correspond to the name of the semaphore, good programming practice suggests that there be some relation.

Like all object references to non-persistent data, when the last reference to a semaphore is reclaimed, the underlying semaphore is also removed.

## 3.4 Semaphore Operations

### 3.4.1 Basic

Semaphore operations can be divided into two major groups: those that manipulate semaphores directly, and those that wait for some other process to manipulate semaphores. The first group consists of:

- Create – Creates a new semaphore instance and initializes it for use
- Open – Accesses and initializes an existing semaphore
- Delete – Makes the semaphore unusable by any process that knows of it
- Increment – Adds a specified amount to the value of a semaphore
- Decrement – If the semaphore value is zero, the operation waits for it to become positive. When the semaphore is positive, the decrement amount (or the value of the semaphore, whichever is less) is subtracted from the semaphore.
- GetValue – Returns the current value of the semaphore
- SetValue – Sets the current value of the semaphore to the non-negative value supplied

### 3.4.2 Managing Multiple Semaphores

The second group of operations involves managing a list of semaphores as a group, and waiting for the operations pending for each on to complete:

- AddToWaitMany – Add the given semaphore operation to the wait list
- RemoveFromWaitMany – Remove the specified semaphore operation from the wait list

- WaitMany – Waits for all the semaphores on the wait list to complete their individual operations. This operation can time out.

### 3.4.3 The Wait List

Each process that coordinates multiple semaphores with **WaitMany** keeps the semaphore decrement operation requests in an internal list. Operations on the list are handled as follows:

- When the **AddToWaitMany** method is called to place a decrement operation in the list, the system attempts to perform the decrement at that time. If the semaphore value is non-zero, the decrement succeeds. The amount decremented is the lesser of the value of the semaphore and the requested amount. Any amont requested greater than the value of the semaphore is forgotten.
- If the semaphore value is zero at the time the request is added to the list, then nothing is done and the request is considered pending. At some time in the future, if the target semaphore becomes non-zero, Caché will select one of the processes with an operation referencing that semaphore and do its decrement operation. If the result of that operation is that the semaphore still has a non-zero value, the Caché will repeat the process until either there are no further requests, or the semaphore value becomes zero.
- When the process calls the **WaitMany** method, Caché examines each of the operations in the wait list. For those that are satisfied, Caché invokes the **WaitCompleted** method of the target semaphore, and then removes the request from the wait list. When it has processed all satisfied requests, it returns the number of such requests to its caller; it returns zero if the wait timeout was exceeded. Requests that were pending and unsatisfied remain on the wait list.
- Because of the asynchronous nature of semaphores, it is possible that one of the pending requests on the wait list may be satisfied during the call to **WaitMany**. Whether this now-satisfied request is counted during this invocation of **WaitMany**, or whether it will be counted on a subsequent invocation is indeterminate.
- It is also possible that a process holding a OREF to a semaphore may delete it. In this case, what happens depends on whether the requested operation was satisfied or not.
  - If the semaphore is the target of an operation that has been satisfied, the request is marked as having decremented the semaphore by zero. The **WaitCompleted** method cannot be called because the semaphore does not exist, but the request is counted as having been satisfied in the value returned by **WaitMany**.
  - If the request is still pending, it is simply removed from the wait list.

### 3.4.4 Callbacks

Instances of semaphores inherit an abstract method, **WaitCompleted**, which users are expected to implement. When processing the satisfied requests on the wait list, **WaitMany** invokes the **WaitCompleted** method on each semaphore in the wait list passing as an argument the amount that the semaphore was decremented. When **WaitCompleted** returns, **WaitMany** removes the request from the wait list.

### 3.4.5 Other Considerations

#### Multiple Decrement Requests On The Same WaitList

It is not an error to request to decrement the same semaphore more than once in the same wait list. The added request is handled as follows:

- If the first request is unsatisfied, the decrement amount from the second request is added to that of the first.
- If the first request has been satisfied (fully or partially), the second request is processed normally. That is, if the semaphore value is non-zero the decrement request is either fully or partially satisfied. However, the actual amount decremented is added to that obtained by the first request.

The decrement amounts are not reported via callback until **WaitMany** is called, so multiple request on the same semaphore appear as if one combined request had been made. This results in scenarios such as:

1. Semaphore A set to 0.
2. Decrement request on A of 4.
3. Decrement request on A of 1; new decrement = 5.
4. Semaphore A set to 4.
5. Request satisfied; 4 granted.
6. Call to waitmany made; WaitCompleted for A reports 4.

1. Semaphore A set to 1.
2. Decrement request on A of 3.
3. Request satisfied; 1 granted.
4. Decrement request on A of 4.
5. Call to WaitMany made; WaitCompleted for A reports 1.

1. Semaphore A set to 1.
2. Decrement request on A of 3.
3. Request satisfied; 1 granted.
4. Decrement request on A of 4.
5. Semaphore A set to 5.
6. Request satisfied; 4 granted. 7. Call to WaitMany made; WaitCompleted for A reports 5; Semaphore A value = 1.

### Semaphore Deletion

Semaphores do not have an owner, and they are not reference-counted as object instances are. Any process that can open a semaphore can delete it.

When a semaphore is deleted,

- If a pending decrement for that semaphore is present in any wait list, the **WaitCompleted** callback is invoked with a decrement value of zero.
- It will be removed from the system (local or remote) where it is mapped.
- An further attempt to access it by any other process will fail with an <INVALID SEMAPHORE> error.

### Job Termination And The Wait List

When a Caché process terminates, its wait list is released. Any satisfied decrement requests that remain in the wait list but have not been processed by **WaitMany** are cleared. Their individual decrement amounts are not added back to the semaphores they decremented. Any unsatisfied requests in the wait list are simply deleted.

### Semaphores And ECP

For a semaphore on an ECP system, operations on a semaphore are sequenced according to the order that the requests arrive at the system holding the semaphore. Each operation is guaranteed to be completed before the next one begins. The following conditions are guaranteed for remote semaphores:

- Semaphore increments and decrements will happen after SETs and KILLs.

- When a semaphore is SET, incremented or decremented, the ECP data cache is coherent with respect to subsequent SETs, increments, or decrements at the server.

Because semaphores are not persistent, in the event of a service interruption, pending semaphore operations across servers on an ECP system are not recoverable. After an ECP outage due to server or network failure, semaphores on the application server are deleted, and pending requests on the data server are deleted. It is the responsibility of the application to detect this condition and recreate the semaphores it needs in their correct state.

# 4 A Simple Producer/Consumer Example

What follows is a series of classes that implement a producer/consumer scenario using a semaphore. A “main” process initializes the semaphore and waits for the user to indicate that the activities are all finished. The producer randomly increments a semaphore value in a loop with a variable delay between updates. The consumer attempts to remove a random amount from the semaphore at random time, also in a loop. The example consists of 5 classes:

- Main – The class that initializes the environment and waits for the activity on the semaphore to complete.
- Counter – The class that implements the semaphore itself. It logs its creation and any callbacks that happen as a result of the semaphore being on a wait list.
- Producer – A class whose main method increments the semaphore value. The increment amount is a randomly chosen small integer. After doing the increment, the method delays a small random number of seconds before the next increment.
- Consumer – This is the complement to Producer. The main method of this class attempts to decrement the semaphore by a randomly chosen small integer. It adds the decrement request to its wait list with a wait time which is also randomly chosen number of seconds.
- Util – This class has several methods used by the other classes of the example. Several methods address the issue of maintaining a common log for all activity; others address the naming of multiple consumers and multiple producers.

The text of the individual classes follows along with a more detailed discussion of their behavior. Afterward, there is a description of how to run the classes along with the variations that are possible.

**Note:** The code that makes up the classes has been purposefully written to be simple. As much as possible, each statement accomplishes only a single action. This should make modification of the example by users easier and more straightforward.

## 4.1 Class: Semaphore.Main

This class establishes the demonstration environment. It invokes the utility class to initialize the log and the name indexing facilities. It then initializes the common semaphore with an initial value of zero and waits for the user to type a character (typically the ENTER key) indicating that the experiment is finished.

Once it receives the user input it reports the current value of the semaphore, attempts to delete it, and terminates execution.

```
/// Environment setup for example
Class Semaphore.Main Extends %RegisteredObject [ ProcedureBlock ]
{

/// The name of the shared semaphore
Parameter ME = "Main";

/// driver for the semaphore demo
ClassMethod Run()
{
    // initialize the logging globals
    Do ##class(Semaphore.Util).InitLog()
    Do ##class(Semaphore.Util).InitIndex()
```

```
    Set msg = ..#ME _ " Started"
    Do ..Log(msg)

    // create and initialize the semaphore
    Set inventory = ##class(Semaphore.Counter).%New()
    If ('($ISOBJECT(inventory))) {
        Set msg = "%New() of MySem failed"
        Do ..Log(msg)
        Quit
    }

    Set msg = "Semaphore create result: " _ inventory.Init(0)
    Do ..Log(msg)

    // wait for termination response
    Set msg = "Enter any character to terminate Run method"
    Do ..Log(msg)

    Read *x

    // report final value, remove the semaphore and finish
    Set msg = "Final value = " _ inventory.GetValue()
    Do ..Log(msg)
    Set msg = "Semaphore delete status: " _ inventory.Delete()
    Do ..Log(msg)
    Set msg = ..#ME _ " Finished"
    Do ..Log(msg)

    Quit
}

/// Enter messages as received into a common log
ClassMethod Log(msg As %String) [ Private ]
{
    Do ##class(Semaphore.Util).Logger(..#ME, msg)
    Quit
}

}
```

## 4.2 Class: Semaphore.Counter

This class implements the semaphore used in the example. As required, it is a subclass of %SYSTEM.Semaphore and provides an implementation for the method, **WaitCompleted**. For the sake of simplicity, the code that initializes the semaphore is also contained in this class. There is also a class method that provides the name of this semaphore to allow it to be obtained by the setup, producer, and consumer classes.

```
/// Local semaphore example class
Class Semaphore.Counter Extends %SYSTEM.Semaphore
{

/// Direct messages to the log facility
ClassMethod Name() As %String
{
    Quit "Counter"
}

/// Direct messages to the log facility
Method Log(Msg As %String) [ Private ]
{
    Do ##class(Semaphore.Util).Logger(..Name(), Msg)
    Quit
}

/// Return the semaphore id value as a hex string
Method MyId() As %String
{
   Quit ("0x" _ $ZHEX(..SemID))
}

/// Invoked when instance created
Method %OnNew() As %Status
{
    Set msg = "New semaphore"
    Do ..Log(msg)
    Quit $$$OK
}
```

```
Method Init(initvalue = 0) As %Status
{
    Try {
        If (..Create(..Name(), initvalue)) {
            Set msg = "Created: """ _ ..Name()
                    _ """; Value = " _ initvalue
                    _ "; Id = 0x" _ ..MyId()
            Do ..Log(msg)
            Return 1
        }
        Else {
        Set msg = "Semaphore create failed: Name = """ _ ..Name() _ """"
        Do ..Log(msg)
        Return 0
        }
    } Catch {
        Set msg = "Semaphore failure caught"
        Do ..Log(msg)
        Return 0
    }
}

Method %OnClose() As %Status [ Private ]
{
    Set msg = "Closing Semaphore: Id = " _ ..MyId()
    Do ..Log(msg)
    Quit $$$OK
}

/// This method is invoked by WaitMany() as a callback.
/// Either a non-zero amount was available in the semaphore or the wait timed out.
/// The amount decremented is passed as the argument to this method; zero, in the
/// case of a timeout.
///
/// After invoking this method, the semaphore is removed from the wait many list.
/// An explicit invocation of AddToWaitMany is required to put it back
/// into the wait list.
Method WaitCompleted(amt As %Integer)
{
    // just report the decrement amount
    Set msg = "WaitCompleted: " _ ..MyId() _ "; Amt = " _ amt
    Do ..Log(msg)

    Quit
}

}
```

## 4.3 Class: Semaphore.Producer

This class is responsible for obtaining an OREF to the common semaphore. Once it has the OREF, it attempts to repeatedly increment the semaphore by a small randomly chosen integer, pausing for a small randomly chosen interval between each increment. Each attempt to increment the semaphore is entered into the log.

```
/// The semaphore increment class
Class Semaphore.Producer Extends %RegisteredObject [ ProcedureBlock ]
{

/// My class name
Parameter MeBase = "Producer";

/// Increments the semaphore by small random amounts after pauses
ClassMethod Run() As %Status
{
    // establish name and access semaphore
    Set ME = ##class(Semaphore.Util).IndexName(..#MeBase)
    Set msg = ME _ " Started"
    Do ..Logger(ME, msg)

    Set cell = ##class(Semaphore.Counter).%New()
    Do cell.Open(##class(Semaphore.Counter).Name())

    Set msg = "Open Id = " _ cell.MyId()
    Do ..Logger(ME, msg)

    // increment semaphore by random amounts
    // at random times
    For addcnt = 1 : 1 : 8 {
        Set incamt = $RANDOM(5) + 1
        Set waitsec = $RANDOM(10) + 1
```

```
        Set msg = "Increment " _ cell.MyId()
                _ " = " _ cell.GetValue()
                _ " by " _ incamt
                _ " wait " _ waitsec _ " sec"
        Do cell.Increment(incamt)
        Do ..Logger(ME, msg)
        Hang waitsec
    }

    // finish
    Set msg = ME _ " Finished"
    Do ..Logger(ME, msg)

    Quit $$$OK
}

/// Channels messages to the central logger
ClassMethod Logger(id As %String, msg As %String) [ Private ]
{
    Do ##class(Semaphore.Util).Logger(id, msg)
    Quit
}

}
```

## 4.4 Class: Semaphore.Consumer

This class is the complement to Semaphore.Producer. It too obtains an OREF to the common semaphore and, in a similar fashion to the Producer, attempts to repeatedly decrement the semaphore by a randomly chosen amount, and with a randomly chosen pause between each attempt. The success or failure of each attempt is written to the log.

```
/// The semaphore decrement class
Class Semaphore.Consumer Extends %RegisteredObject [ ProcedureBlock ]
{

/// My class name
Parameter MeBase = "Consumer";

/// Decrements the semaphore by small random amounts after pauses
ClassMethod Run() As %Status
{
    // establish name and access semaphore
    Set ME = ##class(Semaphore.Util).IndexName(..#MeBase)
    Set msg = ME _ " Started"
    Do ..Logger(ME, msg)

    Set cell = ##class(Semaphore.Counter).%New()
    Do cell.Open(##class(Semaphore.Counter).Name())
    Set msg = "Consumer: Open Id = " _ cell.MyId()
    Do ..Logger(ME, msg)

    // repeatedly decrement the semaphore by
    // variable amounts and varied times
    For deccnt = 1 : 1 : 15 {
        Set decamt = $RANDOM(5) + 1
        Set waitsec = $RANDOM(10) + 1
        Set msg = "Decrement " _ cell.MyId()
                _ " = " _ cell.GetValue()
                _ " by " _ decamt
                _ " wait " _ waitsec _ " sec"
        // in this case we wait for a single semaphore
        // but we could wait on multiple semaphore decrements
        // (up to 200) at one time
        Do cell.AddToWaitMany(decamt)
        Do ..Logger(ME, msg)
        Set result = ##class(%SYSTEM.Semaphore).WaitMany(waitsec)
        Set msg = $SELECT((result > 0):"Granted", 1:"Timeout")
        Do ..Logger(ME, msg)
    }

    // finish
    Set msg = ME _ " Finished"
    Do ..Logger(ME, msg)

    Quit $$$OK
}

/// Channels messages to the central logger
ClassMethod Logger(id As %String, msg As %String) [ Private ]
{
```

```
    Do ##class(Semaphore.Util).Logger(id, msg)
    Quit
}

}
```

## 4.5 Class: Semaphore.Util

This class contains methods that address two issues that relates to this example. The first is the initialization of the structures needed for holding the logged messages together with the methods that both archive the messages submitted to the log and their subsequent display.

The second set of methods address the generation of a numbered sequence of names to identify producers and consumers. This is not strictly needed because the Caché process ids provided by the $JOB command also does this, but it is easier to use more human-readable labels.

```
/// Utility class
Class Semaphore.Util Extends %RegisteredObject [ ProcedureBlock ]
{

/// The name of the shared semaphore
Parameter ME = "Util";

/// initialize output log
ClassMethod InitLog()
{
    // initialize the logging global
    Kill ^SemaphoreLog
    Set ^SemaphoreLog = 0

    Quit
}

/// Enter messages as received into a global
/// for logging purposes
ClassMethod Logger(sender As %String, msg As %String)
{
    Set inx = $INCREMENT(^SemaphoreLog)
    Set ^SemaphoreLog(inx, 0) = $JOB
    Set ^SemaphoreLog(inx, 1) = sender
    Set ^SemaphoreLog(inx, 2) = msg
    Write "(", ^SemaphoreLog, ") ", msg, !
    Quit
}

/// display the messages in the log
ClassMethod ShowLog()
{
    Set msgcnt = $GET(^SemaphoreLog, 0)
    Write "Message Log: Entries = ", msgcnt, !, !
    Write "#", ?5, "$JOB", ?12, "Sender", ?25, "Message", !

    For i = 1 : 1 : msgcnt {
        Set job = ^SemaphoreLog(i, 0)
        Set sender = ^SemaphoreLog(i, 1)
        Set msg = ^SemaphoreLog(i, 2)
        Write i, ")", ?5, job, ?12, sender, ":", ?25, msg, !
    }
    Quit
}

/// initialize the name index
ClassMethod InitIndex()
{
    // initialize the logging global
    Kill ^SemaphoreNames

    Quit
}

/// initialize the name index
ClassMethod IndexName(name As %String) As %String
{
    If ($DATA(^SemaphoreNames(name)) = 0) {
        Set ^SemaphoreNames(name) = 0
    }

    Set index =  $INCREMENT(^SemaphoreNames(name))
```

```
    Quit (name _ "." _ index)
}

}
```

# 4.6 Running The Examples

## 4.6.1 Overview

Each of the three classes Main, Producer, and Consumer has its own **Run** method and is best exercised by running each of them in its own terminal window. As each runs, it will display the messages it generates for the log. Once the user has responded to the Main class by supplying the input it is waiting for, the **Run** method of Main will terminate removing the semaphore. The user can then see a display of the consolidated log file from all processes by typing the command

```
Do ##class(Semaphore.Util).ShowLog()
```

**Note:** All the following examples assume that all the classes have been compiled in the “USER” namespace.

## 4.6.2 Example 1 — Create And Remove the Semaphore

The simplest example demonstrates the creation and destruction of the semaphore. It uses the Semaphore.Main class. Do the following:

1. Open a terminal window.
2. Type the command –

   ```
   Do ##class(Semaphore.Main).Run()
   ```

3. The method will attempt to create the semaphore. If it is successful, you will see the message, “Enter any character to terminate Run method”. Depress the Enter key. The method displays the initialized value of the semaphore, deletes it, and exits.
4. Display the log file by issuing the command,

   ```
   Do ##class(Semaphore.Util).ShowLog()
   ```

An example of the messages displayed in the terminal window from following the above steps is

```
USER>Do ##class(Semaphore.Main).Run()
(1) Main Started
(2) New semaphore
(3) Created: "Counter"; Value = 0; Id = 0x0x10000
(4) Semaphore create result: 1
(5) Enter any character to terminate Run method
<ENTER>
(6) Final value = 0
(7) Semaphore delete status: 1
(8) Main Finished
(9) Closing Semaphore: Id = 0x10000
```

And the log output looks like this:

```
USER>Do ##class(Semaphore.Util).ShowLog()
Message Log: Entries = 9

#    $JOB   Sender       Message
1)   7176   Main:        Main Started
2)   7176   Counter:     New semaphore
3)   7176   Counter:     Created: "Counter"; Value = 0; Id = 0x0x10000
4)   7176   Main:        Semaphore create result: 1
5)   7176   Main:        Enter any character to terminate Run method
6)   7176   Main:        Final value = 0
7)   7176   Main:        Semaphore delete status: 1
8)   7176   Main:        Main Finished
9)   7176   Counter:     Closing Semaphore: Id = 0x10000
```

### 4.6.3 Example 2 — Create the Semaphore And Successively Increment It

This example shows the Producer at work, as well as the capture of log messages from both processes.

1. Open two separate terminal windows. Call them “A” and “B”.
2. In window A, type the following command, but do not type the ENTER key at the end –

   ```
   Do ##class(Semaphore.Main).Run()
   ```

3. In window B, type the following command, but again, do not type the ENTER key at the end of the command –

   ```
   Do ##class(Semaphore.Producer).Run()
   ```

4. Now, in window A, press the ENTER key. Then in window B, press the ENTER key. This will start both classes executing in parallel. Their individual messages are displayed in their own window.
5. When the Producer process finishes, close the B window.
6. In the A window, pressing the ENTER key to allow the Main class to finish. Then, display the log with the command –

   ```
   Do ##class(Semaphore.Util).ShowLog()
   ```

For this example, the following were the outputs

Window A

```
USER>Do ##class(Semaphore.Main).Run()
(1) Main Started
(2) New semaphore
(3) Created: "Counter"; Value = 0; Id = 0x0x30002
(4) Semaphore create result: 1
(5) Enter any character to terminate Run method
<ENTER>
(19) Final value = 28
(20) Semaphore delete status: 1
(21) Main Finished
(22) Closing Semaphore: Id = 0x30002
```

Window B

```
USER>Do ##class(Semaphore.Producer).Run()
(6) Producer.1 Started
(7) New semaphore
(8) Open Id = 0x30002
(9) Increment 0x30002 = 0 by 5 wait 3 sec
(10) Increment 0x30002 = 5 by 2 wait 7 sec
(11) Increment 0x30002 = 7 by 2 wait 8 sec
(12) Increment 0x30002 = 9 by 4 wait 1 sec
(13) Increment 0x30002 = 13 by 5 wait 6 sec
(14) Increment 0x30002 = 18 by 3 wait 3 sec
(15) Increment 0x30002 = 21 by 4 wait 4 sec
(16) Increment 0x30002 = 25 by 3 wait 3 sec
(17) Producer.1 Finished
(18) Closing Semaphore: Id = 0x30002
```

Log display

```
USER>Do ##class(Semaphore.Util).ShowLog()
Message Log: Entries = 22

#    $JOB   Sender       Message
1)   7628   Main:        Main Started
2)   7628   Counter:     New semaphore
3)   7628   Counter:     Created: "Counter"; Value = 0; Id = 0x0x30002
4)   7628   Main:        Semaphore create result: 1
5)   7628   Main:        Enter any character to terminate Run method
6)   9036   Producer.1:  Producer.1 Started
7)   9036   Counter:     New semaphore
8)   9036   Producer.1:  Open Id = 0x30002
9)   9036   Producer.1:  Increment 0x30002 = 0 by 5 wait 3 sec
10)  9036   Producer.1:  Increment 0x30002 = 5 by 2 wait 7 sec
```

```
11)  9036   Producer.1:  Increment 0x30002 = 7 by 2 wait 8 sec
12)  9036   Producer.1:  Increment 0x30002 = 9 by 4 wait 1 sec
13)  9036   Producer.1:  Increment 0x30002 = 13 by 5 wait 6 sec
14)  9036   Producer.1:  Increment 0x30002 = 18 by 3 wait 3 sec
15)  9036   Producer.1:  Increment 0x30002 = 21 by 4 wait 4 sec
16)  9036   Producer.1:  Increment 0x30002 = 25 by 3 wait 3 sec
17)  9036   Producer.1:  Producer.1 Finished
18)  9036   Counter:     Closing Semaphore: Id = 0x30002
19)  7628   Main:        Final value = 28
20)  7628   Main:        Semaphore delete status: 1
21)  7628   Main:        Main Finished
22)  7628   Counter:     Closing Semaphore: Id = 0x30002
```

### 4.6.4 Example 3 — Run All Three Processes Together

This example shows attempts to increment and decrement the same semaphore in a coherent manner. It users all three major classes.

1. Open three separate terminal windows. Call them “A”, “B”, and “C”.
2. In window A, type the following command, but do not press the ENTER key at the end –

   ```
   Do ##class(Semaphore.Main).Run()
   ```

3. In window B, type the following command, but again, do not press the ENTER key at the end of the command –

   ```
   Do ##class(Semaphore.Producer).Run()
   ```

4. In window C, type the following command, but again, do not press the ENTER key at the end of the command –

   ```
   Do ##class(Semaphore.Consumer).Run()
   ```

5. Starting with window A, visit each window and type the ENTER key. This will start the Main class and then each of other two classes. As previously, each process will display its log messages in its own window.
6. When both the Producer and Consumer processes finish, close the B window and the C window.
7. In the A window, press the ENTER key to allow the Main class to finish. Then, display the log with the command –

   ```
   Do ##class(Semaphore.Util).ShowLog()
   ```

Running this example produces output similar to the following:

Window A

```
USER>Do ##class(Semaphore.Main).Run()
(1) Main Started
(2) New semaphore
(3) Created: "Counter"; Value = 0; Id = 0x0x60005
(4) Semaphore create result: 1
(5) Enter any character to terminate Run method
<ENTER>
(65) Final value = 0
(66) Semaphore delete status: 1
(67) Main Finished
(68) Closing Semaphore: Id = 0x60005
```

Window B

```
USER>Do ##class(Semaphore.Producer).Run()
(6) Producer.1 Started
(7) New semaphore
(8) Open Id = 0x60005
(9) Increment 0x60005 = 0 by 2 wait 3 sec
(17) Increment 0x60005 = 0 by 5 wait 5 sec
(24) Increment 0x60005 = 0 by 4 wait 10 sec
(38) Increment 0x60005 = 0 by 2 wait 7 sec
(42) Increment 0x60005 = 0 by 1 wait 8 sec
(48) Increment 0x60005 = 0 by 1 wait 10 sec
(54) Increment 0x60005 = 0 by 5 wait 7 sec
(58) Increment 0x60005 = 0 by 1 wait 7 sec
(62) Producer.1 Finished
(63) Closing Semaphore: Id = 0x60005
```

Window C

```
USER>Do ##class(Semaphore.Consumer).Run()
(10) Consumer.1 Started
(11) New semaphore
(12) Consumer: Open Id = 0x60005
(13) Decrement 0x60005 = 2 by 4 wait 1 sec
(14) WaitCompleted: 0x60005; Amt = 2
(15) Granted
(16) Decrement 0x60005 = 0 by 1 wait 4 sec
(18) WaitCompleted: 0x60005; Amt = 1
(19) Granted
(20) Decrement 0x60005 = 4 by 4 wait 6 sec
(21) WaitCompleted: 0x60005; Amt = 4
(22) Granted
(23) Decrement 0x60005 = 0 by 1 wait 9 sec
(25) WaitCompleted: 0x60005; Amt = 1
(26) Granted
(27) Decrement 0x60005 = 3 by 2 wait 4 sec
(28) WaitCompleted: 0x60005; Amt = 2
(29) Granted
(30) Decrement 0x60005 = 1 by 4 wait 4 sec
(31) WaitCompleted: 0x60005; Amt = 1
(32) Granted
(33) Decrement 0x60005 = 0 by 3 wait 1 sec
(34) Timeout
(35) Decrement 0x60005 = 0 by 1 wait 7 sec
(36) Timeout
(37) Decrement 0x60005 = 0 by 5 wait 7 sec
(39) WaitCompleted: 0x60005; Amt = 2
(40) Granted
(41) Decrement 0x60005 = 0 by 2 wait 8 sec
(43) WaitCompleted: 0x60005; Amt = 1
(44) Granted
(45) Decrement 0x60005 = 0 by 4 wait 2 sec
(46) Timeout
(47) Decrement 0x60005 = 0 by 4 wait 7 sec
(49) WaitCompleted: 0x60005; Amt = 1
(50) Granted
(51) Decrement 0x60005 = 0 by 5 wait 9 sec
(52) Timeout
(53) Decrement 0x60005 = 0 by 5 wait 9 sec
(55) WaitCompleted: 0x60005; Amt = 5
(56) Granted
(57) Decrement 0x60005 = 0 by 1 wait 9 sec
(59) WaitCompleted: 0x60005; Amt = 1
(60) Granted
(61) Consumer.1 Finished
```

Log display

```
USER>Do ##class(Semaphore.Util).ShowLog()
Message Log: Entries = 68

#    $JOB   Sender       Message
1)   3976   Main:        Main Started
2)   3976   Counter:     New semaphore
3)   3976   Counter:     Created: "Counter"; Value = 0; Id = 0x0x60005
4)   3976   Main:        Semaphore create result: 1
5)   3976   Main:        Enter any character to terminate Run method
6)   4852   Producer.1:  Producer.1 Started
7)   4852   Counter:     New semaphore
8)   4852   Producer.1:  Open Id = 0x60005
9)   4852   Producer.1:  Increment 0x60005 = 0 by 2 wait 3 sec
10)  6128   Consumer.1:  Consumer.1 Started
11)  6128   Counter:     New semaphore
12)  6128   Consumer.1:  Consumer: Open Id = 0x60005
13)  6128   Consumer.1:  Decrement 0x60005 = 2 by 4 wait 1 sec
14)  6128   Counter:     WaitCompleted: 0x60005; Amt = 2
15)  6128   Consumer.1:  Granted
16)  6128   Consumer.1:  Decrement 0x60005 = 0 by 1 wait 4 sec
17)  4852   Producer.1:  Increment 0x60005 = 0 by 5 wait 5 sec
18)  6128   Counter:     WaitCompleted: 0x60005; Amt = 1
19)  6128   Consumer.1:  Granted
20)  6128   Consumer.1:  Decrement 0x60005 = 4 by 4 wait 6 sec
21)  6128   Counter:     WaitCompleted: 0x60005; Amt = 4
22)  6128   Consumer.1:  Granted
23)  6128   Consumer.1:  Decrement 0x60005 = 0 by 1 wait 9 sec
24)  4852   Producer.1:  Increment 0x60005 = 0 by 4 wait 10 sec
25)  6128   Counter:     WaitCompleted: 0x60005; Amt = 1
26)  6128   Consumer.1:  Granted
27)  6128   Consumer.1:  Decrement 0x60005 = 3 by 2 wait 4 sec
28)  6128   Counter:     WaitCompleted: 0x60005; Amt = 2
```

```
29)  6128   Consumer.1:  Granted
30)  6128   Consumer.1:  Decrement 0x60005 = 1 by 4 wait 4 sec
31)  6128   Counter:     WaitCompleted: 0x60005; Amt = 1
32)  6128   Consumer.1:  Granted
33)  6128   Consumer.1:  Decrement 0x60005 = 0 by 3 wait 1 sec
34)  6128   Consumer.1:  Timeout
35)  6128   Consumer.1:  Decrement 0x60005 = 0 by 1 wait 7 sec
36)  6128   Consumer.1:  Timeout
37)  6128   Consumer.1:  Decrement 0x60005 = 0 by 5 wait 7 sec
38)  4852   Producer.1:  Increment 0x60005 = 0 by 2 wait 7 sec
39)  6128   Counter:     WaitCompleted: 0x60005; Amt = 2
40)  6128   Consumer.1:  Granted
41)  6128   Consumer.1:  Decrement 0x60005 = 0 by 2 wait 8 sec
42)  4852   Producer.1:  Increment 0x60005 = 0 by 1 wait 8 sec
43)  6128   Counter:     WaitCompleted: 0x60005; Amt = 1
44)  6128   Consumer.1:  Granted
45)  6128   Consumer.1:  Decrement 0x60005 = 0 by 4 wait 2 sec
46)  6128   Consumer.1:  Timeout
47)  6128   Consumer.1:  Decrement 0x60005 = 0 by 4 wait 7 sec
48)  4852   Producer.1:  Increment 0x60005 = 0 by 1 wait 10 sec
49)  6128   Counter:     WaitCompleted: 0x60005; Amt = 1
50)  6128   Consumer.1:  Granted
51)  6128   Consumer.1:  Decrement 0x60005 = 0 by 5 wait 9 sec
52)  6128   Consumer.1:  Timeout
53)  6128   Consumer.1:  Decrement 0x60005 = 0 by 5 wait 9 sec
54)  4852   Producer.1:  Increment 0x60005 = 0 by 5 wait 7 sec
55)  6128   Counter:     WaitCompleted: 0x60005; Amt = 5
56)  6128   Consumer.1:  Granted
57)  6128   Consumer.1:  Decrement 0x60005 = 0 by 1 wait 9 sec
58)  4852   Producer.1:  Increment 0x60005 = 0 by 1 wait 7 sec
59)  6128   Counter:     WaitCompleted: 0x60005; Amt = 1
60)  6128   Consumer.1:  Granted
61)  6128   Consumer.1:  Consumer.1 Finished
62)  4852   Producer.1:  Producer.1 Finished
63)  4852   Counter:     Closing Semaphore: Id = 0x60005
64)  6128   Counter:     Closing Semaphore: Id = 0x60005
65)  3976   Main:        Final value = 0
66)  3976   Main:        Semaphore delete status: 1
67)  3976   Main:        Main Finished
68)  3976   Counter:     Closing Semaphore: Id = 0x60005
```

### 4.6.5 Other Variations

Other variations on this example are possible. Although there can be only one Semaphore.Main running at one time, there is no restriction on the number of Producers or Consumers executing in other windows. Users are encouraged to try differing numbers of consumer and producers in various scenarios such as

- Try running three consumers and a single producer so there is greater “competition” for the semaphore. With luck, the log will show that two or more consumers made requests to decrement the semaphore and both succeeded because the semaphore value was large enough to satisfy some or all of both requests.
- You can also use these classes to demonstrate what happens in other processes when the semaphore is deleted. To do this, while Producers or Consumers are running, switch to the window where the Main class is running and press ENTER. In the course of finishing its processing, the Main class will delete the semaphore and the OREF of the Producer or Consumer will no longer be valid. The next attempt to use will generate a error.
- By changing the name of the semaphore to one which looks like a global name, you can have the semaphore mapped to a different instance on, for example, an ECP system.