Article
Iryna Mykhailova · Sep 25, 2018 12m read

Implementing concurrent access to shared resources using Semaphores

If you ' ve ever wondered whether there is a way to regulate access to resources in Caché, wonder no more. In version 2014.2 special classes were added that allow developers to work with semaphores. Basically, a semaphore is a non-negative integer variable that can be affected by two types of operations:

- P-operation on a semaphore is an attempt to decrease the value of the semaphore by 1. If the value of the semaphore is greater than 1 prior to the execution of the P-operation, the P-operation is performed without a delay. If the value of the semaphore before the operation is 0, the process performing the P-operation is switched to a waiting state until the value becomes greater than 0;
- V-operation on a semaphore is an increment of its value by 1. If there are processes that are delayed during the execution of the P-operation on this semaphore, one of these processes leaves the waiting state and can perform its own P-operation.

There are two types of semaphores: binary and counting. The difference is that in the former case a variable can only assume two values (0 and 1), and in the latter – any non-negative integer number.

Following are several use cases where semaphores come in handy.

1. Mutual exclusion on a semaphore

To implement a mutual exception – for example, to prevent simultaneous modification of shared data by two or more processes, a binary semaphore S is created. The initial value of this semaphore is 1. Critical sections (those that can be executed only by one process at a time) are framed by P(S) (at the beginning) and V(S) (at the end) brackets. A process, which enters a critical section, performs a P(S) operation and switches the semaphore to 0. If there is another process in a critical section, and consequently the value of the semaphore is already 0, this process is blocked in its P-operation until the current process finishes and performs its V(S) operation upon quitting.

2. Synchronization on a semaphore

For synchronization purposes, one creates a binary semaphore S with the initial value of 0. The initial 0 value means that the event hasn ' t occurred yet. The process that signals about an event performs the V(S) operation that sets the value to 1. The process waiting for an event performs the P(S) operation. If the event has already occurred by that time, the waiting process continues to run. If not, the process switches to the waiting state until the signaling process performs V(S) operation.

If there are several processes waiting for the same event, the process that successfully performs the P(S) operation must immediately run the V(S) operation to send another event signal for the next queued process.

3. A semaphore resource counter

If you have N units of a particular resource, a general S semaphore with a value of N is created for controlling its allocation. Resources are allocated with the P(S) command and released with the V(S) command. The value of the semaphore, therefore, reflects the number of free resource units. If the value of the semaphore is 0, which means that there are no more units available, any process requesting this resource will be switched to the waiting state in

the P(S) operation until one of the processes using the resource releases it by performing the V(S) operation.

All these options can be implemented in Caché using the %SYSTEM.Semaphore class, which encapsulates a 64-bit non-negative integer number and provides methods for changing its value for all active processes.

Let ' s take a look at how a semaphore counter can be used in a sample situation (third use case).

Let ' s assume that a university has 10 slots for accessing an international database of scientific articles. This way, you have a resource that needs to be shared among students willing to have access. Each student has their own login/password for accessing the database, but no more than 10 users from the university can work with the system simultaneously. Each time a student attempts to log in to the database, we need to decrease the number of available slots by one. Accordingly, when the student leaves, we need to return this one slot to the general access pool.

In order to check whether a particular student should be granted access or be queued, we are going to use a semaphore counter with an initial value of 10. To see how the permission granting process works, we use logging. The main class will initialize variables and monitor the work of the students. Another class will be inherited from %SYSTEM.Semaphore and will create a semaphore. Let ' s also create a separate class for various utilities. And the last two classes will emulate a student ' s sign-in and sign-out from the database. Since the server " knows " the name of the user signing into the system and leaving it, we ' ll create a separate global for simulating this " knowledge " that will store information about active users (^LoggedUsers). We will use it to register the logins of students signing in and picking random names for those signing out.

Let ' s start with the Main class where we will do the following:

1. Create a semaphore,
2. Set it to the initial value (equal to 10, which corresponds to 10 free slots for accessing the scientific articles database),
3. Stop the process and remove the semaphore,
4. Show the log.

```
Class SemaphoreSample.Main Extends %RegisteredObject [ ProcedureBlock ]
{

/// sample driver
ClassMethod Run()
{
    // initializing globals for logging
    Do ##class(SemaphoreSample.Util).InitLog()
    Do ##class(SemaphoreSample.Util).InitUsers()

    Set msg = "Process start "
    Do ..Log(msg)

    // creating and initializing a semaphore
    Set inventory = ##class(SemaphoreSample.Counter).%New()
    If ('($ISOBJECT(inventory))) {
        Set msg = "The SemaphoreSample.Counter %New() class method hasn't worked yet"
        Do ..Log(msg)
        Quit
    }

    // setting the initial semaphore value
    if 'inventory.Init(10) {
        Set msg = "There was a problem initializing the semaphore"
        Do ..Log(msg)
        Quit
        }
```

```
    // waiting for the process to end
    Set msg = "Press any key to block access..."
    Do ..Log(msg)

    Read *x

    //Removing the semaphore
    Set msg = "The semaphore has been removed with the following status: " _ inventor
y.Delete()
    Do ..Log(msg)
    Set msg = " Process end "
    Do ..Log(msg)

    do ##class(SemaphoreSample.Util).ShowLog()

    Quit
}

/// Calling a utility for writing a log
ClassMethod Log(msg As %String) [ Private ]
{
    Do ##class(SemaphoreSample.Util).Logger($Horolog, "Main", msg)
    Quit
}

}
```

The next class that we will create is a class with various utilities. They will be required for our test application to work. It will contain class methods responsible for:

1. Preparing the logging global for operation (^SemaphoreLog),
2. Saving logs to the global,
3. Showing logs,
4. Saving the names of active users to the global (^LoggedUsers),
5. Selecting a random name from the list of active users,
6. Removing a name of the active users global by index.

```
Class SemaphoreSample.Util Extends %RegisteredObject [ ProcedureBlock ]
{

/// initializing the log
ClassMethod InitLog()
{
    // removing previous records from the log
    Kill ^SemaphoreLog
    Set ^SemaphoreLog = 0

    Quit
}

/// initializing the log
ClassMethod InitUsers()
{
    //removing all users from the global just in case
        if $data(^LoggedUsers) '= 0
    {
        Kill ^LoggedUsers
```

```
    }
    Set ^LoggedUsers = 0
}

/// writing the log to the global
ClassMethod Logger(time As %DateTime, sender As %String, msg As %String)
{
    Set inx = $INCREMENT(^SemaphoreLog)
    Set ^SemaphoreLog(inx, 0) = time
    Set ^SemaphoreLog(inx, 1) = sender
    Set ^SemaphoreLog(inx, 2) = msg
    Write "(", ^SemaphoreLog, ") ", msg_" ? "_$ztime($PIECE(time,",",2), 1), !
    Quit
}

/// showing messages on screen
ClassMethod ShowLog()
{
    Set msgcnt = $GET(^SemaphoreLog, 0)
    Write "Message log: number of records = ", msgcnt, !, !
    Write "#", ?5, "Time", ?12, "Sender", ?25, "Message", !

    For i = 1 : 1 : msgcnt {
        Set time = ^SemaphoreLog(i, 0)
        Set sender = ^SemaphoreLog(i, 1)
        Set msg = ^SemaphoreLog(i, 2)
        Write i, ")", ?5, $ztime($PIECE(time,",",2), 1), ?15, sender, ":", ?35, msg,
!
    }
    Quit
}

/// adding a user to the list of signed-in users
ClassMethod AddUser(Name As %String)
{
    Set inx = $INCREMENT(^LoggedUsers)
    set ^LoggedUsers(inx) = Name
}

/// removing a user from the list of signed-in users
ClassMethod DeleteUser(inx As %Integer)
{
    kill ^LoggedUsers(inx)
}

/// selecting a user name from the list of signed-in users
ClassMethod ChooseUser(ByRef Name As %String) As %Integer
{
    // if all users are “out”, we return the need to wait sign
        if $data(^LoggedUsers) = 1
    {
       Set Name = ""
       Quit -1
    } else
    {
        Set Temp = ""
        Set Numb = $Random(10)+5
           For i = 1 : 1: Numb {
                 Set Temp = $Order(^LoggedUsers(Temp))
```

```
                    // to limit the loop to one level of the global only
                    // we move the pointer to the beginning after each pass
                  if (Temp = "")
                  {
                 set Temp = $Order(^LoggedUsers(""))
                  }
            }
            set Name = ^LoggedUsers(Temp)
            Quit Temp
        }
}

}
```

Next class implements a semaphore. It extends the %SYSTEM.Semaphore system class and consists of the following methods which:

1. Return the semaphore ' s unique name,
2. Save events to the log,
3. Log the event of creation and destruction the semaphore (callback methods),
4. Create and initialize the semaphore.

```
Class SemaphoreSample.Counter Extends %SYSTEM.Semaphore
{

/// Each counter must have a unique name
ClassMethod Name() As %String
{
    Quit "Counter"
}

/// Calling a utility to write to the log
Method Log(Msg As %String) [ Private ]
{
    Do ##class(SemaphoreSample.Util).Logger($Horolog, ..Name(), Msg)
    Quit
}

/// Callback method for creating a new object
Method %OnNew() As %Status
{
    Set msg = "Creating a semaphore "
    Do ..Log(msg)
    Quit $$$OK
}

/// Creating and initializing a semaphore
Method Init(initvalue = 0) As %Status
{
    Try {
        If (..Create(..Name(), initvalue)) {
            Set msg = "Created: """ _ ..Name()
                    _ """; Initial value = " _ initvalue
            Do ..Log(msg)
            Return 1
        }
        Else {
        Set msg = "There was a problem creating a semaphore with the name = """ _ ..N
```

```
ame() _ """"
        Do ..Log(msg)
        Return 0
        }
    } Catch errobj {
        Set msg = "There was an error creating a semaphore: "_errobj.Data
        Do ..Log(msg)
        Return 0
    }
}

/// Callback method used for closing an object
Method %OnClose() As %Status [ Private ]
{
    Set msg = "Closing the semaphore "
    Do ..Log(msg)
    Quit $$$OK
}

}
```

It is important to know that semaphore is identified by a name that is passed to the system when it is created. This name must comply with the requirements for local/global variables and be unique. As a rule, semaphores are stored in the database instance where it was created, and is visible for other processes of this instance. If the name of a semaphore meets the naming requirements for global variables, it becomes available to all active processes, including ECP.

The last two classes simulate users signing in and out of the system. For simplicity ’ s sake, let ’ s assume that there are exactly 25 users. Of course, we could start the process and keep creating a new user until a key is pressed, but I decided to keep it simple and use a finite loop. In both classes, we connect to an existing semaphore and try to decrease (sign in)/increase (sign out) the counter. We assume that the student will wait for his turn for an indefinite period of time (he really needs to get into this library), so we use the Decrement function that allows us to set an infinite wait period. Otherwise, we can specify an exact timeout period in tenths of a second. To prevent all users from trying to sign in simultaneously, lets add an arbitrary pause before log-in attempts.

```
Class SemaphoreSample.LogIn Extends %RegisteredObject [ ProcedureBlock ]
{

/// Simulating user sign-ins to the system
ClassMethod Run() As %Status
{

    //opening a semaphore responsible for access to the database
    Set cell = ##class(SemaphoreSample.Counter).%New()
    Do cell.Open(##class(SemaphoreSample.Counter).Name())

    // starting to log in to the system
    // let’s take 25 different students for this example
    For deccnt = 1 : 1 : 25 {
        // generating a random login
       Set Name = ##class(%Library.PopulateUtils).LastName()

        try
        {
           Set result =  cell.Decrement(1, -1)
        } catch
        {
           Set msg = "Access blocked"
```

```
            Do ..Logger(##class(SemaphoreSample.Counter).Name(), msg)
            Return
        }
        do ##class(SemaphoreSample.Util).AddUser(Name)
        Set msg = Name _ " entered the system "
        Do ..Logger(Name, msg)

        Set waitsec = $RANDOM(10) + 7
        Hang waitsec
    }
    Set msg = "There are no more users waiting to log in to the system"
    Do ..Logger(##class(SemaphoreSample.Counter).Name(), msg)
    Quit $$$OK
}

/// Calling a utility for saving a log
ClassMethod Logger(id As %String, msg As %String) [ Private ]
{
    Do ##class(SemaphoreSample.Util).Logger($Horolog, id, msg)
    Quit
}

}
```

In our model, when we get disconnected from the server, we should also check if there are other still connected users. Therefore, we will first look at the content of the global containing users (^LoggedUsers), and, if it ' s empty, wait for some time, then check if anyone managed to sign in.

```
Class SemaphoreSample.LogOut Extends %RegisteredObject [ ProcedureBlock ]
{

/// Simulating user log-outs
ClassMethod Run() As %Status
{
    Set cell = ##class(SemaphoreSample.Counter).%New()
    Do cell.Open(##class(SemaphoreSample.Counter).Name())

    // logging out of the system
    For addcnt = 1 : 1 : 25 {
        Set inx = ##class(SemaphoreSample.Util).ChooseUser(.Name)
        while inx = -1
        {
            Set waitsec = $RANDOM(10) + 1
            Hang waitsec
            Set inx = ##class(SemaphoreSample.Util).ChooseUser(.Name)
        }
        try
        {
            Do cell.Increment(1)
        } catch
        {
            Set msg = "Access blocked"
            Do ..Logger(##class(SemaphoreSample.Counter).Name(), msg)
            Return
        }


        Set waitsec = $RANDOM(15) + 2
```

```
            Hang waitsec
        }
        Set msg = "All users have logged out of the system"
        Do ..Logger(##class(SemaphoreSample.Counter).Name(), msg)
        Quit $$$OK
    }

    /// Calling a utility to saving the log
    ClassMethod Logger(id As %String, msg As %String) [ Private ]
    {
        Do ##class(SemaphoreSample.Util).Logger($Horolog, id, msg)
        Quit
    }

    }
```

The project is ready. Let ' s compile it so that we can launch it and see the result. We ' ll do it in three separate Terminal windows.

In the first window, if needed, switch to the necessary namespace (I worked on the project in the USER namespace)

```
zn "USER"
```

and call the method that initiates the launch of our " server " from the Main class:

```
do ##class(SemaphoreSample.Main).Run()
```

In the second window, we call the Run method from the Login class that will generate users logging in to the system:

```
do ##class(SemaphoreSample.LogIn).Run()
```

In the last window, we call the Run method from the LogOut class that will generate users logging out of the system:

```
do ##class(SemaphoreSample.LogOut).Run()
```

Once everyone has logged in and out, you will have logs in all three windows. For a clearer picture, it ' s better to wait for the first 10 authorized users to log in to the system to demonstrate that the value of the semaphore cannot be less than 0 before starting the LogOut routine.

Apart from Increment and Decrement methods used in this example, you can use the waiting list and corresponding methods to work with the semaphore:

- AddToWaitMany — add a semaphore-related operation to the list
- RmFromWaitMany — remove an operation from the list
- WaitMany — wait until all operations with the semaphore are over

In this case, WaitMany loops through all the tasks on the list and, once the operation is successfully completed, calls the WaitCompleted method that the developer needs to implement on his own. It is called when the semaphore has assigned a non-zero value for an operation or the wait has ended due to a timeout. The number by

which the counter is decreased is returned to an argument of this method (0 in case of a timeout). Once the method has finished working, the semaphore is removed from the WaitMany task list and the next task is taken.

More information about semaphores in Caché can be found in the official documentation (it also features another example of a semaphore) and in the class description.

The project is hosted on GitHub.

Your comments and suggestions are highly appreciated!

#Best Practices #Tips & Tricks #Tutorial #Caché
