Multi-Threaded Programming Guide

SoftVelocity

SoftVelocity Incorporated
2769 East Atlantic Blvd.
Pompano Beach, Florida 33062
(954) 785-4555
www.softvelocity.com

Printed in the United States of America (0903)

Contents:

Overview

This paper introduces programming techniques you implement when you program in a preemptive multi-threading environment as introduced in Clarion 6.0. It identifies the type of code and objects that require attention, and also provides code examples for adding thread synchronization to properly handle access to shared non-threaded data.

Throughout this paper there are references to static variables. There are three ways of creating a variable in Clarion such that it is static:

- ☐ The variable is declared in Global Data and does not have the THREAD attribute
- ☐ The variable is declared in Module Data and does not have the THREAD attribute
- ☐ The variable is declared in Procedure Data and has the STATIC attribute, but does not have the THREAD attribute

Static variables are potentially dangerous in a preemptive environment, and should be avoided where possible. Later sections in this paper will describe what to do when they cannot be avoided.

What is a threading model?

A threading model describes how different threads work together under a multi-threading operating system. There are two types of threading models: cooperative threading and preemptive threading.

Under the cooperative threading model an application could have multiple threads ‘running’ at the same time. But only one of these threads could do anything at any point in time. For another thread (or program) to get a chance to do anything the current thread had to relinquish control of the PC back to the operating system. The operating system would then choose which thread would next have a chance to do something. Thus the threads cooperate with each other and the operating system to give each other a chance to do something. This was easy to program, but it meant that programs could behave badly and take full control of the PC locking out other programs.

When Windows NT arrived so did a new style of threading: Preemptive multi-threading. Now threads could run simultaneously, either on separate CPUs or by having the operating system suspend one thread and start another whenever the operating system wanted. This released the power of multi-processor machines and stopped the problem of badly behaved programs. But it was a much harder paradigm to program with.

Threads Pre-Clarion 6

Clarion for Windows was designed to make programming easy. Therefore the problems of preemptive threading were considered unnecessarily complex for the rewards. Clarion threads were not cooperative by their nature. They were standard Win32 threads with explicit synchronization implemented in the RTL to emulate cooperative behavior.

The emulation of cooperative threads made it easy to use variables that were global in scope (can be used anywhere in your program), but whose contents were different depending on what thread was active; for example a global variable with the THREAD attribute. To do this some mechanism was required to make sure that the contents changed each time a different thread became active. Due to the cooperative nature of Clarion threads the RTL was responsible to swap the contents of the threaded variables every time a thread gained or lost focus.

This made it very easy to program, but added the restriction that only one Clarion thread could be active at any time

Threads in Clarion 6

Clarion 6 now supports the preemptive thread model. To do this, the automatic data swapping that was done by the RTL had to end. In Clarion 6, the operating system determines when a thread receives a CPU time slice. So there is no opportunity for the RTL to swap the contents of variables.

Although Clarion now supports preemptive multi-threading, it does not mean you have to suddenly rewrite all your programs. If you follow the guidelines set out in this paper for avoiding the use of global variables whenever possible and using the synchronization techniques described, you’ll be able to deliver new capabilities to your end-users.

Code differences between Clarion 5.5 and Clarion 6

To allow Clarion to support preemptive threads some changes were required in the language. This was kept to a minimum so as to ease the migration to the new version.

THREAD attribute on Classes

Prior to Clarion 5.5 if you had a global variable that was an instance of a class the THREAD attribute was ignored. In Clarion 6 any global instance of a class with the THREAD attribute will have separate data for each running thread. The constructor of the class will be called on thread initialization and the destructor will be called on thread termination.

Thread “initialization" and thread “termination" must be defined because it is not always obvious. In general, any DLL has some entry point called by the OS on attaching that DLL to the process, detaching it from the process, on the start and closing a thread. The Clarion RTL translates every such call, to chains of calls to initialize or cleanup data in all modules which have application-wide or thread-wide variables declared. But in certain situations the OS does not do calls to executables, for example, if it is an EXE rather than DLL, or if the DLL is loaded dynamically by the call to LoadLibrary. In the latter case, the OS calls that DLL's entry point for initializing application-wide data only. The CW RTL handles all special cases and calls the code to initialize and also to cleanup threaded data.

The order that class constructors are called is undefined. So do not assume that a different class instance has been constructed when your constructor is called

This feature gives you a lot of power to control things on a thread by thread basis. The ABC templates use this strategy to make sure the meta-information about your database is available on each thread.

See the section below “Using hand code” for an example of a global threaded class.

THREAD and EXTERNAL

In versions of Clarion prior to version 6, a variable with the THREAD attribute was allocated a fixed memory location and the runtime library handled swapping the thread specific values of that variable in and out of that memory location. So if you had a file definition in a DLL you would declare it as

```
AFile FILE,DRIVER(‘TopSpeed’),THREAD
```

Then in your EXE you would declare the file as

```
AFile FILE,DRIVER(‘TopSpeed’),EXTERNAL
```

With Clarion 6.0 a variable with the THREAD attribute has different memory allocated for each thread. The compiler now generates code to make sure that the right memory is accessed regardless of which thread is running. For the compiler to generate the right code it needs to know if a variable has the THREAD attribute regardless of where the variable is defined. So in Clarion 6.0 if you have a file definition in a DLL you declare it as

```
AFile FILE,DRIVER(‘TopSpeed’),THREAD
```

Then in your EXE you declare the file as

```
AFile FILE,DRIVER(‘TopSpeed’),EXTERNAL, THREAD
```

Notice the THREAD attribute is present in both declarations, unlike previous versions.

ADDRESS of a threaded variable

Before taking the ADDRESS() of a threaded variable would always return the same result regardless of what thread is running. In Clarion 6.0 every thread has its own memory location for a threaded variable. So ADDRESS() no longer will return the same value. If you need a constant address for a threaded global variable you can use the new function INSTANCE() and pass a Thread Number of 0. For example, to get a unique identifier for a threaded FILE that you can be certain will be the same regardless of what thread you are on you used to do ADDRESS(MyFile). You now do INSTANCE(MyFile, 0)

LOCKTHREAD, UNLOCKTHREAD and THREADLOCKED

As Clarion 6 now works with preemptive threads rather than cooperative threads these functions do nothing by default. However, if you want to work in a cooperative threading environment (see next paper) LOCKTHREAD will call the SYSTEM{PROP:LockThreadHook} function, UNLOCKTHREAD will call the SYSTEM{PROP:UnlockThreadHook} function and THREADLOCKED will call the SYSTEM{PROP:ThreadLockedHook} function.

Coding techniques for Preemptive multi-threading

Don’t Use Static Variables

The easiest way to avoid problems is to avoid using static variables. A lot of static variables are declared as static simply because it took less time to use a global or module variable then to pass the data as a parameter. Check your static variables, and if they are used to transfer information between threads pass them as a parameter as needed.

Initialize Once/Read Many

Non Queue static variables

If you have static variables that you use to hold global information that never changes during the running of a program, then you do not need to change them. Provided the variables are initialized in the startup of the program and never changed, there is no need to worry. For example, prior to Clarion 6, when there was no SYSTEM{PROP:DataPath}, it was common to have a file name initialized based on a path stored in an INI file. This practice is still acceptable in a preemptive threading model.

Static Queues that are initialized at startup

A static queue that is initialized at startup and read many times is **not safe**. See the section below on Common Coding Practices that need to change for ideas on how to handle static queues.

Synchronize Access

If you are using static variables as a way to share information between threads, then you will need to synchronize thread access to these variables. The two main ways you will do this is either using a Critical Section Object or a ReaderWriterLocks Object.

If the value to be protected is a LONG (and not in a GROUP) you are probably safe in not synchronizing most thread accesses, since reads/writes happen in only one operation. If a LONG is defined in a GROUP/RECORD/QUEUE then it might not be aligned data and require more then one operation. In summary a LONG has fewer risks of corrupt half updated data then strings, but still have some risks when multiple commands are involved.

See the Synchronization Objects section below for descriptions and examples on how to use these objects.

IMPORTANT!

It is very important that you lock out other threads for as short a time as possible. Ideally, copying the data you need into thread safe variables then releasing the lock. You should not have any user input inside a piece of code that locks other threads.

To guarantee that your static variables are protected it is best to move them into a static class that has read and write methods for manipulating them. Make them PRIVATE, create Get and Set methods, and have synchronization code to protect and manipulate them.

Common Coding Practices that need to change

Using static variables to pass parameters

If you are using a static variable to pass values to a started procedure on a new thread, change the prototype of that procedure so the values it requires are passed as parameters. Both the ABC and Clarion template chains now support this feature.

Using static queues

Static queues that are shared amongst multiple threads will require you to make some code changes with regard to how they are accessed to work in a preemptive threading environment. The problem with queues is that they not only have a queue buffer, but they also have position information. So if one thread reads from the first element of the queue and another thread reads from the second element of the queue you could end up with the following situation:

Thread 1	**Thread 2**
GET(Queue, 1)	
	GET(Queue, 2)
	Assign Queue Data to local var
Assign Queue Data to local var	

Thread 1 gets a time slice from the OS and reads from the Queue, then the OS give thread 2 a time slice before thread 1 makes the assignment of the Queue data to a local variable. Now both thread 1 and thread 2 ends up reading the information for element 2 of the Queue. To avoid this situation you need to synchronize access to the queue’s buffer. To keep the time that other threads are locked out while the queue is accessed to a minimum, you should read from the queue and assign the data from the queue’s buffer into a threaded memory buffer. See the section below on ICriticalSection for a complete example showing how to do this.

Queues that are only modified at startup

A common coding technique is to set up a static queue that contains data that does not change, for example the states of the USA. This is initialized on program startup, and used throughout the application.

This technique needs to be adjusted to work in a preemptive environment for the reasons stated above. Luckily, the solution is easy -- put a THREAD attribute on the QUEUE

This means that every thread will have its own instance of the QUEUE. The trouble is you need to get the data into that queue for every thread. To accomplish this you create a threaded class to do the copying for you. Below is an example class that will populate a queue for every thread.

In the following example QLock is used to make sure that two threads starting up simultaneously will not interfere with each other. The initial population of the Queue does not need modification provided it is done on the main program thread before any other threads are started. The example shown also makes use of the new INSTANCE() language statement.

```
GlobalQ QUEUE,THREAD
Data     STRING(10)
        END

QLock &ICriticalSection
GlobalQueuePopulator CLASS,THREAD
Construct             PROCEDURE
Destruct              PROCEDURE
                     END

GlobalQueuePopulator.Construct PROCEDURE
BaseQ &GlobalQ
recs  UNSIGNED,AUTO
i     UNSIGNED,AUTO

  CODE
    IF THREAD() <> 1
      QLock.Wait()
      BaseQ &= INSTANCE(GlobalQ, 1)
      recs = RECORDS(BaseQ)
      LOOP i = 1 TO recs
        GET(BaseQ, i)
        GlobalQ.Data = BaseQ.Data
        ADD(GlobalQ)
      END
      QLock.Release()
    ELSE
      QLock &= NewCriticalSection()
    END

GlobalQueuePopulator.Destruct PROCEDURE
  CODE
    IF THREAD() = 1
      QLock.Kill()
    END
```

The WAIT() function

The WAIT function will wait until no other threads want the object. It then takes hold of the object until a subsequent call to Release. Other threads that call the WAIT function will wait indefinitely until the other thread releases the object.

Proper programming techniques are essential here to avoiding a “deadly embrace”, or deadlock.

For example, if you execute the following:

```
SyncObj1.Wait
SyncObj2.Wait
```

on one thread and

```
SyncObj2.Wait
SyncObj1.Wait
```

on another thread, you are risking a deadlock.

Deadlocks should not be a big issue if you make sure you have a hierarchy of synchronization objects. Ideally, you only use one object at a time, but if you must use multiple objects, always acquire a lock on the top synchronization object first, then the other one. That way, you can never get into the aforementioned scenario.

Thread Synchronization

The Clarion runtime has a variety of built in interfaces and procedures to help you maintain synchronization between your threads. The POST and EVENT functions have always been in the Clarion language for thread synchronization. SUSPEND and RESUME functions allow you to stop and start another thread, INSTANCE allows you to get another thread's contents for a variable and the ICriticalSection, IMutex, ISemaphore and IReaderWriterLock are interfaces to objects that allow you to synchronize the processing between multiple threads and also multiple processes.

POST/EVENT

You have always been able to synchronize thread processing by posting an event from one thread to another using POST() to send the event and EVENT() to receive it. See the SUSPEND/RESUME section below for an example on using these functions to synchronize two threads.

SUSPEND/RESUME

SUSPEND allows you to stop another process. RESUME starts that process again. You can issue multiple SUSPEND calls for a thread. The same number of RESUME calls must be made for that thread to restart.

The SUSPEND procedure suspends a thread specified by the threadno parameter. If the threadno parameter is a number of an active thread, its execution is suspended and a suspending counter is incremented. Each additional SUSPEND statement issued to the same active thread will increment the suspending counter by one. Therefore, a thread that has been suspended with a given number of SUSPEND statements can only resume thread execution when an equal number of RESUME statements has been executed.

EXTREME CAUTION should be taken with MDI programs using SUSPEND, as improper use can cause program lockups. All MDI child windows have an MDI client window as a parent, and the MDI client window can send rather than post messages to its child windows.

For example, calling the inter-thread SendMessage modal function causes the calling thread (the MDI client window) to suspend activity until the called thread (the MDI Child window) returns from the call. If the called thread is suspended, we would have a program lockup.

The SUSPEND and RESUME functions can be very useful for controlling threads that are CPU intensive. For example, rebuilding keys on a file. Here is an example program that starts a BUILD of a file and allows the user to pause the build and restart it.

```
    PROGRAM
    MAP
      DoBuild(STRING)
    END

MyFile FILE,DRIVER('TopSpeed'),PRE(F)
Key1     KEY(F:Field1),PRIMARY
Key2     KEY(F:Field2)
Key3     KEY(F:Field3, Field4)
         RECORD
Field1     LONG
Field2     STRING(20)
Field3     STRING(20)
Field4     STRING(20)
         END
       END
```

```
BuilderWin WINDOW('Building File'),AT(,,81,22),GRAY
             BUTTON('Suspend Build'),AT(2,3,75,14),USE(?Button)
           END

AllDone  EQUATE(500H)
Building BYTE
ThreadID SIGNED,AUTO

  CODE
    OPEN(BuilderWin)
    ThreadID = START(DoBuild, , THREAD())
    Building = TRUE
    ACCEPT
      CASE EVENT()
      OF AllDone
        MESSAGE('Build Complete')
        BREAK
      OF Event:Accepted
        IF ACCEPTED() = ?Button
          IF Building
            SUSPEND(ThreadID)
            ?Button{PROP:Text} = 'Resume Building'
          ELSE
            RESUME(ThreadID)
            ?Button{PROP:Text} = 'Suspend Build'
          END
        END
      END
    END

DoBuild PROCEDURE (parent)
  CODE
    MyFile{PROP:FullBuild} = TRUE
    BUILD(MyFile)
    POST(AllDone,,parent)
```

INSTANCE

In versions of Clarion prior to Clarion 6.0 a variable’s memory location was constant regardless of which thread accessed the variable. Therefore this code would always work:

```
  PROGRAM
  MAP
    AFunc()
  END

GlobVar SIGNED,THREAD
Addr    LONG

  CODE
    Addr = ADDRESS(GlobVar,1)
    START(AFunc)

AFunc PROCEDURE
  CODE
    IF Addr <> ADDRESS(GlobVar)
      MESSAGE('Panic')
    END
```

This sort of code was used in ABFILE.CLW to make sure the file manager matched the file it was meant to manage. To allow programs to know what variable they are really using you can now use the INSTANCE function to get the address of the variable on any thread, and most importantly on thread 1. The above code would need to be modified as follows to work in Clarion 6.0.

```
  PROGRAM
  MAP
    AFunc()
  END

GlobVar SIGNED,THREAD
Addr    LONG

  CODE
    Addr = ADDRESS(GlobVar)
    START(AFunc)

AFunc PROCEDURE
  CODE
    IF Addr <> INSTANCE(GlobVar,0)
      MESSAGE('Panic')
    END
```

Synchronization Objects

A Synchronization Object is an object used to control how multiple threads cooperate in a preemptive environment. There are four Synchronization Objects supplied by the Clarion runtime: Critical Sections (ICriticalSection), Mutexes (IMutex), Semaphores (ISemaphore), and Read/Write Locking (IReaderWriterLock).

WARNING:

> *Due to the fact that Windows uses procedure-modal methods when dealing with MDI based applications (a program with an APPLICATION window) you must not have any user input when you have control of a synchronization object with an MDI based application. This is likely to lead to your application locking up*
>
> *If you must have user input, then you must release control of the synchronization object while waiting for user input.*

IcriticalSection and CriticalSection

You use an ICriticalSection when you want only one thread accessing some resource (e.g. a global, non-threaded variable) at any one time. An ICriticalSection is faster than an IMutex. If you do not need the extra features of an IMutex, use an IcriticalSection

CriticalSection is a built-in class that allows for easy creation of simple, global synchronization objects.

Following are a couple of examples that make sure that only one thread is accessing a static queue at a time.

```
  PROGRAM

! This program assumes that only WriteToQueue and
! ReadFromQueue directly access NonThreadedQueue
! If other code accesses the queue and does not use
! the QueueLock critical section to synchronize
! access to the queue, then all the work inside WriteToQueue
! and ReadFromQueue is wasted

QueueData GROUP,THREAD
ThreadID    LONG
Information STRING(20)
          END

  !Include CriticalSection
  INCLUDE(‘CWSYNCHC.INC’),ONCE
  MAP
    WriteToQueue()
    WriteToQueue(*QueueData)
    ReadFromQueue()
    ReadFromQueue(*QueueData)
  END

NonThreadedQueue QUEUE
Data               LIKE(QueueData)
                 END
```

```
QueueLock CriticalSection

  CODE

    ! Do everything


WriteToQueue PROCEDURE()
! Assumes QueueData is used to pass data.  This is thread safe
! because QueueData has the THREAD attribute
  CODE
    QueueLock.Wait()  !Lock access to NonThreadedQueue.
    NonThreadedQueue.Data = QueueData
    GET(NonThreadedQueue, NonThreadedQueue.Data.ThreadId)
    IF ERRORCODE()
      ADD(NonThreadedQueue)
    ELSE
      PUT(NonThreadedQueue)
    END
    QueueLock.Release() !Allow other access to the queue

WriteToQueue PROCEDURE(*QueueData in)
  CODE
    QueueLock.Wait()  !Lock access to NonThreadedQueue.
    NonThreadedQueue.Data = in
    GET(NonThreadedQueue, NonThreadedQueue.Data.ThreadId)
    IF ERRORCODE()
      ADD(NonThreadedQueue)
    ELSE
      PUT(NonThreadedQueue)
    END
    QueueLock.Release() !Allow other access to the queue

ReadFromQueue PROCEDURE()
! Returns results in QueueData.  This is thread safe
! because QueueData has the THREAD attribute
  CODE
    QueueLock.Wait()  !Lock access to NonThreadedQueue.
    NonThreadedQueue.Data.ThreadId = THREAD()
    GET(NonThreadedQueue, NonThreadedQueue.Data.ThreadId)
    QueueData = NonThreadedQueue.Data
    QueueLock.Release() !Allow other access to the queue

ReadFromQueue PROCEDURE(*QueueData out)
  CODE
    QueueLock.Wait()  !Lock access to NonThreadedQueue.
    NonThreadedQueue.Data.ThreadId = THREAD()
    GET(NonThreadedQueue, NonThreadedQueue.Data.ThreadId)
    out = NonThreadedQueue.Data
    QueueLock.Release() !Allow other access to the queue
```

The previous example suffers from the problem that anyone can access the global queue and they are not forced to use the matching QueueLock critical section. The following example removes this problem by moving the non threaded queue and the critical section into a static class.

```
  PROGRAM


QueueData GROUP,THREAD
ThreadID    LONG
Information STRING(20)
          END

NonThreadedQueue QUEUE,TYPE
Data               LIKE(QueueData)
                 END

QueueAccess CLASS
QueueData &NonThreadedQueue,PRIVATE
QueueLock &ICriticalSection,PRIVATE
Construct     PROCEDURE
Destruct      PROCEDURE
WriteToQueue  PROCEDURE
WriteToQueue  PROCEDURE(*QueueData)
ReadFromQueue PROCEDURE()
ReadFromQueue PROCEDURE(*QueueData)
            END

  INCLUDE(‘CWSYNCHC.INC’)
  MAP
  END

  CODE
    ! Do everything

QueueAccess.Construct PROCEDURE
  CODE
    SELF.QueueLock &= NewCriticalSection()
    SELF.QueueData &= NEW(NonThreadedQueue)

QueueAccess.Destruct PROCEDURE
  CODE
    SELF.QueueLock.Kill()
    DISPOSE(SELF.QueueData)

QueueAccess.WriteToQueue PROCEDURE()
! Assumes QueueData is used to pass data. This is thread safe
! because QueueData has the THREAD attribute
  CODE
    SELF.QueueLock.Wait()  !Lock access to NonThreadedQueue.
    SELF.QueueData.Data = QueueData
    GET(SELF.QueueData, SELF.QueueData.Data.ThreadId)
    IF ERRORCODE()
      ADD(SELF.QueueData)
    ELSE
      PUT(SELF.QueueData)
    END
    SELF.QueueLock.Release() !Allow other access to the queue

QueueAccess.WriteToQueue PROCEDURE(*QueueData in)
  CODE
    SELF.QueueLock.Wait()  !Lock access to NonThreadedQueue.
    SELF.QueueData.Data = in
    GET(SELF.QueueData, SELF.QueueData.Data.ThreadId)
    IF ERRORCODE()
```

```
      ADD(SELF.QueueData)
    ELSE
      PUT(SELF.QueueData)
    END
    SELF.QueueLock.Release() !Allow other access to the queue

QueueAccess.ReadFromQueue PROCEDURE()
! Returns results in QueueData.  This is thread safe
! because QueueData has the THREAD attribute
  CODE
    SELF.QueueLock.Wait()  !Lock access to NonThreadedQueue.
    SELF.QueueData.Data.ThreadId = THREAD()
    GET(SELF.QueueData, SELF.QueueData.Data.ThreadId)
    QueueData = SELF.QueueData.Data
 SELF.QueueLock.Release() !Allow other access to the queue

QueueAccess.ReadFromQueue PROCEDURE(*QueueData out)
  CODE
   SELF.QueueLock.Wait()  !Lock access to NonThreadedQueue.
   SELF.QueueData.Data.ThreadId = THREAD()
   GET(SELF.QueueData, SELF.QueueData.Data.ThreadId)
   out = SELF.QueueData.Data
   SELF.QueueLock.Release() !Allow other access to the queue
```

IWaitableSyncObject

The IWaitableSyncObject is the base interface for IMutex and ISemaphore. This allows you to create procedures that work with either type of synchronization object without requiring the procedure to know exactly what type of object it is.

IMutex

An IMutex is used when you want to allow only one thread to access a resource. Just like an ICriticalSection. However, IMutexes have the added features of being able to not only synchronize threads, but also synchronize different processes. Thus, if you have a resource that can only have one process accessing it at one time (e.g. a registration file that controls access to multiple programs) then you will need to use an IMutex that is created by calling NewMutex(*Name*). *Name* must be the same for all processes that use it to access the same
set of shared resources.

Another time you would use an IMutex rather than an ICriticalSection is if you do not want to always lock a thread.

Finally, an IMutex works better than an ICriticalSection in MDI applications as ICriticalSection objects may cause deadlocks.

A Mutex is a very simple way to limit the user to having one instance of your program running at any time. The following example shows the use of the Name parameter for creating a Mutex and the TryWait method to limit your program in this way.

```
PROGRAM
 INCLUDE('CWSYNCHM.INC'),ONCE
 MAP
 END

Limiter &IMutex,AUTO
Result  SIGNED,AUTO
LastErr LONG,AUTO          !<< return error

  CODE
  Limiter &= NewMutex('MyApplicationLimiterMutex',,LastErr)
  IF Limiter &= NULL
   MESSAGE ('ERROR: Mutex can not be created ' & LastErr)
  ELSE
   Result = Limiter.TryWait(50)
    IF Result <= WAIT:OK
     !Do Everything
     Limiter.Release()     !release
    ELSIF Result = WAIT:TIMEOUT
     MESSAGE('Timeout')
    ELSE
     MESSAGE('Waiting is failed ' & Result) !show Result
    END
   Limiter.Kill()
  END
```

The difference between an IMutex and an ISemaphore is an IMutex can only have one thread successfully wait. An ISemaphore can have multiple threads successfully wait. It is also possible to create a semaphore where no thread can successfully wait.

ISemaphore with multiple successful waits

An ISemaphore created with an initial thread count other than zero will allow you to call Wait that number of times before the wait will lock. For example a semaphore created with `MySem &= NewSemaphore(,2)` will allow `MySem.Wait()` to succeed twice without any call to `MySem.Release()`. This is a very easy way to limit the number of threads you have active at any one time.

If at any time you want to allow more calls to Wait to succeed, you can make additional Release() calls. The number of extra threads that can be added in this way is limited by the final parameter of NewSemaphore(). If you do not want to allow this feature, do not specify the final a maximum.

See the following section for a semaphore that limits the number of threads of a specific type to one.

ISemaphore with no waits

A semaphore created with an initial thread count of 0 will block any call to Wait until a Release is called. You use this type of semaphore to signal another thread that they can do something. For example, signal a thread to send an email to someone because you have sold the last candy bar.

Following is an example where the no wait style of semaphore is used to signal a reader that there is something to read. A multiple successful waits semaphore is used to limit the number of reader threads to 1.

```
   PROGRAM

    INCLUDE('CWSYNCHM.INC')
    INCLUDE('CWSYNCHC.INC')
    INCLUDE('ERRORS.CLW')
    MAP
      Reader()
      Writer()
    END

LogFile FILE,DRIVER('ASCII'),CREATE,NAME('LogFile.txt'),THREAD
          RECORD
Line        STRING(255)
          END
        END

AccessToGlobals CriticalSection
NewData         Semaphore
LimitReaders    &ISemaphore

GlobalStrings   QUEUE,PRE(Q)
Data              STRING(50)
                END

AppFrame APPLICATION('Reader/Writer'),AT(,,400,240),SYSTEM,MAX,RESIZE
       MENUBAR
         MENU('&File'),USE(?FileMenu)
           ITEM('E&xit'),USE(?Exit),STD(STD:Close)
         END
         MENU('&Launch'),USE(?LaunchMenu)
           ITEM('Reader'),USE(?LaunchReader)
           ITEM('Writer'),USE(?LaunchWriter)
         END
       END
     END

  CODE
    LimitReaders &= NewSemaphore(1)
    SHARE(LogFile)
    IF ERRORCODE() = NoFileErr
      CREATE(LogFile)
      SHARE(LogFile)
    END
    IF ERRORCODE()
      STOP('Log File could not be opened.  Error: ' & ERROR())
    END
    OPEN(AppFrame)
    ACCEPT
      IF EVENT() = EVENT:Accepted
        CASE ACCEPTED()
        OF ?LaunchReader
          START(Reader)
        OF ?LaunchWriter
```

```
            START(Writer)
          END
        END
      END
      ! Test to see if Reader is still alive
      IF LimitReaders.TryWait(1) = WAIT:TIMEOUT
        !It is, so lets kill it
        AccessToGlobals.Wait()
        Q:Data = 'exit'
        ADD(GlobalStrings)
        AccessToGlobals.Release()
        NewData.Release() ! Release the reader that is waiting
        LimitReaders.Wait() ! Wait for thread to terminate
        LimitReaders.Release()
      END
      LimitReaders.Kill()

  Reader PROCEDURE
    CODE
      ! Check that there are no other readers
      IF LimitReaders.TryWait(1) = WAIT:TIMEOUT
        MESSAGE('Only One Reader Allowed at a time.  ' &|
                'Kill reader by typing ''exit'' in a sender')
        RETURN
      END
      ! If TryWait succeeds, then we have control of
      ! the LimitReaders Semaphore
      SHARE(LogFile)
  l1  LOOP
        NewData.Wait() !Wait until a writer signals that there is some data
        AccessToGlobals.Wait()
        LOOP
          GET(GlobalStrings,1)
          IF ERRORCODE() THEN BREAK.
          IF Q:Data = 'exit' THEN
            FREE(GlobalStrings)
            AccessToGlobals.Release()
            BREAK l1
          ELSE
            LogFile.Line = Q:Data
            ADD(LogFile)
          END
          DELETE(GlobalStrings)
        END
        AccessToGlobals.Release()
      END
      CLOSE(LogFile)
      LimitReaders.Release() !Allow a new Reader
      RETURN
```

```
Writer PROCEDURE

LocString STRING(50)
Window WINDOW('Writer'),AT(,,143,43),GRAY
       PROMPT('Enter String'),AT(2,9),USE(?Prompt1)
       ENTRY(@s50),AT(45,9,95,10),USE(LocString)
       BUTTON('Send'),AT(2,25,45,14),USE(?Send)
       BUTTON('Close'),AT(95,25,45,14),USE(?Close)
     END

  CODE
    OPEN(Window)
    Window{PROP:Text} = 'Writer ' & THREAD()
    ACCEPT
      IF EVENT() = EVENT:Accepted
        CASE ACCEPTED()
        OF ?Send
          AccessToGlobals.Wait()
          Q:Data = LocString
          ADD(GlobalStrings)
          AccessToGlobals.Release()
          NewData.Release() ! Release the reader that is waiting
        OF ?Close
          BREAK
        END
      END
    END
    RETURN
```

IReaderWriterLock

An IReaderWriterLock can be used to allow multiple threads to read from a global resource, but for only one thread to write to it. No reader is allowed to read the resource if someone is writing and no one can write to the resource if anyone is reading.

An example of this would be where the user specifies screen colors. This is stored in an INI file and read on startup. As the user can change these at any time, the code that changes the values needs to obtain a write lock and all those that read the color information need to obtain a read lock.

Static queues cannot be synchronized via an IReaderWriterLock because reading a queue also modifies its position. All queue access must be considered to cause writes.

Here is some simple code that reads and writes to some static variables. To make sure that no one accesses the statics outside the locking mechanism, the variables are declared as PRIVATE members of a static class.

```
    PROGRAM

    INCLUDE('CWSYNCHM.INC'),ONCE
    MAP
    END

GlobalVars CLASS
AccessToGlobals     &IReaderWriterLock,PRIVATE
BackgroundColor     LONG,PRIVATE
TextSize            SHORT,PRIVATE
Construct           PROCEDURE
Destruct            PROCEDURE
GetBackground       PROCEDURE(),LONG
PutBackground       PROCEDURE(LONG)
GetTextSize         PROCEDURE(),SHORT
PutTextSize         PROCEDURE(SHORT)
           END
```

```
  CODE

GlobalVars.Construct           PROCEDURE
  CODE
    SELF.AccessToGlobals &= NewReaderWriterLock()

GlobalVars.Destruct            PROCEDURE
  CODE
    SELF.AccessToGlobals.Kill()

GlobalVars.GetBackground PROCEDURE()
ret SHORT,AUTO
Reader &ISyncObject
  CODE
    ! You need to copy the static variable
    ! to somewhere safe (A local variable) which
    ! can then be returned without fear that
    ! another thread will change it
    Reader &= SELF.AccessToGlobals.Reader()
    Reader.Wait()
    ret = SELF.Background
    Reader.Release()
    RETURN ret

GlobalVars.PutBackground PROCEDURE(SHORT newVal)
Writer &ISyncObject
  CODE
    Writer &= SELF.AccessToGlobals.Writer()
    Writer.Wait()
    SELF.Background = newVal
    Writer.Release()

GlobalVars.GetTextSize         PROCEDURE()
ret SHORT,AUTO
Reader &ISyncObject
  CODE
    ! You need to copy the static variable
    ! to somewhere safe (A local variable) which
    ! can then be returned without fear that
    ! another thread will change it
    Reader &= SELF.AccessToGlobals.Reader()
    Reader.Wait()
    ret = SELF.TextSize
    Reader.Release()
    RETURN ret

GlobalVars.PutTextSize         PROCEDURE(SHORT newVal)
Writer &ISyncObject
  CODE
    Writer &= SELF.AccessToGlobals.Writer()
    Writer.Wait()
    SELF.TextSize = newVal
    Writer.Release()
```

CriticalProcedure

The CriticalProcedure class is a very easy way to use an ISyncObject interface. If you create a local instance of a CriticalProcedure and initialize it, then it will look after the waiting for a lock and releasing the lock on the ISyncObject for you. The main advantage of using the CriticalProcedure class to handle the locking and releasing for you is that if you have multiple RETURN statements in your procedure, you do not have to worry about releasing the lock before each one. The destructor of the CriticalProcedure will handle that for you.

For example, the following code

```
    PROGRAM
    MAP
     WRITETOFILE()
    END

    INCLUDE('CWSYNCHM.INC')

ERRORFILE FILE,DRIVER('ASCII'),PRE(EF)
RECORD     RECORD
LINE        STRING(100)
           END
          END

LOCKER &ICRITICALSECTION

    CODE
    Locker &= NewCriticalSection()
    ASSERT(~Locker &= Null)
    !do everything

WriteToFile PROCEDURE()
  CODE
    Locker.Wait()
    OPEN(ErrorFile)
    IF ERRORCODE()
      Locker.Release()
      RETURN
    END
    SET(ErrorFile)
    NEXT(ErrorFile)
    IF ERRORCODE()
      Locker.Release()
      RETURN
    END
    EF:Line = ‘Something’
    PUT(ErrorFile)
    CLOSE(ErrorFile)
    Locker.Release()
```

Can be shortened, and made less error prone, to this:

```
    PROGRAM
    MAP
     WRITETOFILE()
    END

    INCLUDE('CWSYNCHM.INC')

ERRORFILE FILE,DRIVER('ASCII'),PRE(EF)
RECORD     RECORD
LINE        STRING(12)
           END
          END

LOCKER &ICRITICALSECTION

    CODE
    Locker &= NewCriticalSection()
    ASSERT(~Locker &= Null)
    !do everything

WriteToFile PROCEDURE()
```

```
CP CriticalProcedure

  CODE
    CP.Init(Locker)
    OPEN(ErrorFile)
    IF ERRORCODE()
      RETURN
    END
    SET(ErrorFile)
    NEXT(ErrorFile)
    IF ERRORCODE()
      RETURN
    END
    EF:Line = ‘Something’
    PUT(ErrorFile)
    CLOSE(ErrorFile)
```

Summary of Synchronization Objects Properties and Methods

The following section provides a quick summary description of the common properties and methods used with the Synchronization Objects discussed previously.

Prototypes

Note: The following prototypes can be found in **CWSYNCHM.INC**

Get Mutex

GetMutex(*name*, **<***error***>**)

Purpose:
Returns a reference to an IMutex for a Mutex that has already been created with NewMutex( )

name	A string constant or variable that names the **IMutex** object.
err	A LONG variable that returns any operating system error.

If the Mutex has not been previously created, a NULL is returned.

Example:

```
MySem    &IMutex
   CODE
   MySem &= GetMutex('MyApp_RequestServer')
   IF MySem &= NULL
     MESSAGE('Server not running')
     RETURN(False)
   END
```

Get Semaphore

GetSemaphore(*name*, **<***error***>**)

Purpose:
Returns a reference to an ISemaphore for a semaphore that has already been created with NewNamedSemaphore( )

name	A string constant or variable that names the **ISemaphore** object.
err	A LONG variable that returns any operating system error.

If the semaphore has not been previously created, a NULL is returned.

Example:

```
MySem    &ISemaphore
   CODE
   MySem &= GetSemaphore('MyApp_RequestServer')
   IF MySem &= NULL
       MESSAGE('Server not running')
      RETURN(False)

   END
```

NewCriticalSection

NewCriticalSection ()

Purpose:
Returns a reference to a new ICriticalSection

NewMutex

NewMutex ( ),
NewMutex (*name*, *owner, <error>*)

Description:
Returns a reference to a new IMutex. If the Mutex could not be created, a Null value will be returned. Check the *Error* parameter for the reason. (Some reasons for failure can be that another object (e.g., semaphore) exists with the same name, or a different user has created the object (e.g., security error)).

Purpose:

name	A string constant or variable that names the new IMutex object. If a *name* parameter is not supplied, **NewMutex** is used for synchronizing threads only. If a *name* is supplied, NewMutex can be used to synchronize multiple processes rather than just the threads within a process.
owner	A BYTE variable or constant that specifies the initial owner of the Mutex. If this value is TRUE, and the caller creates the Mutex, the calling thread obtains ownership of the Mutex. This is the equivalent of calling **Mutex.Wait** immediately after **NewMutex**(). If the caller did not create the mutex, then the calling thread does not obtain ownership of the Mutex. To determine if the caller created the Mutex, you need to check the value of *Err*
error	A LONG variable or constant that returns any operating system error. Err = ERROR_ALREADY_EXISTS (183) indicates that the caller did not create the Mutex (a handle is still returned but owner is ignored) because another process or thread has already created the Mutex.

NewNamedSemaphore

NewNamedSemaphore (*name*, *initial* , *max* , <*error*>)

name	A string constant or variable that names the new **ISemaphore** object.
initial	A LONG variable or constant that indicates the maximum number of threads that can access the semaphore. The default value is zero.
max	A LONG variable or constant that indicates the maximum number of threads that can simultaneously access the semaphore.
error	A LONG variable that returns any operating system error. An ERROR_ALREADY_EXISTS (183) error indicates that another process or thread has already created the Mutex.

Purpose:
Returns a reference to a new ISemaphore. This semaphore can be used for synchronizing threads or processes. If an ERROR_ALREADY_EXISTS (183) is posted to *error*, then the Semaphore already existed, and *initial* and *max* are ignored.

Using the default settings, NewNamedSemaphore will create a semaphore that no threads can wait on until someone calls a Release.

NewReaderWriterLock

NewReaderWriterLock (*WritersHavePriority*)

Purpose:
Returns a reference to a new IreaderWriterLock

WritersHavePrioruty	A BYTE variable or constant that sets priority of the Writer. If *WritersHavePriority* is TRUE, a Writer waiting for ownership of the ReaderWriterLock object will take priority over any Readers that are waiting for ownership. If *WritersHavePriority* is FALSE, all Readers waiting for ownership of the ReaderWriterLock object will take priority over any Writer waiting for ownership.

NewSemaphore

NewSemaphore (*initial* , *max*)

Purpose:
Returns a reference to a new ISemaphore

initial	A LONG variable or constant that indicates the maximum number of threads that can access the semaphore. The default value is zero.
max	A LONG variable or constant that indicates the maximum number of times that the semaphore can be owned. It is a maximum resource count.

If initial is non-zero, then this indicates how many threads can initially and simultaneously have access to the semaphore

max indicates the maximum number of threads that can simultaneously access the semaphore. By default, this will create a semaphore that no threads can wait on until a Release is called.

Properties and Methods Summary

ICriticalSection.Wait( ) and IMutex.Wait( )

Implementation:

The first time either method is called, the calling thread gains control of the synchronization object. Any other threads that call Wait will wait until **Release** is called.

A thread can call **Wait** multiple times. **Release**() must be called once for each call to **Wait**.

ICriticalSection.Release( ) and IMutex.Release(*count*)

Implementation:
Each method decrements the counter of **Wait** calls. When the **Wait** count reaches zero, the synchronization object is released for any other thread to acquire control with a call to **Wait**.

Count specifies to release a given number of times up to the number of previously successful wait calls.

Kill( )

Implementation:
Releases all resources allocated when the target synchronization object was created with the appropriate **NEW** function.

ISemaphore.Wait( )

Implementation:

This method will wait until shared control of the synchronization object is allowed. Once this occurs, Wait will take partial control of the synchronization object.

A thread can call **Wait( )** multiple times. **Release( )** must be called once for each call to Wait.

ISemaphore.TryWait (ms) and IMutex.TryWait (ms)

```
TryWait          PROCEDURE(LONG milliseconds),SIGNED,PROC
```

Implementation:

Attempts to gain control of the synchronization object within the *ms* parameter time in milliseconds. Return value is a signed EQUATE with one of the following results:

WAIT:OK
Control was acquired

WAIT:TIMEOUT
Control could not be acquired within the time limit

WAIT:NOHANDLE
Something is seriously wrong

WAIT:FAILED
Something else is seriously wrong

WAIT:ABANDONED
Another thread that had control of the synchronization object has ended without calling release. The caller now has the control the same way as if a WAIT:OK was returned. This is more of a warning indicating a flaw in the program logic.

ISemaphore.Release( )

Implementation:
Decrements the counter of **Wait()** calls. When the **Wait()** count reaches zero, the synchronization object is released for any other thread to acquire control with a call to **Wait()**.

If more **Release()** calls are made than **Wait()** calls, the total number of threads allowed to control the semaphore is incremented up to the maximum parameter of *NewSemaphore*.

Count specifies to release a given number of times up to the number of previously successful wait calls.

ISemaphore.Handle( ) and IMutex.Handle( )

For internal use only

IReaderWriterLock.Reader( )

Purpose:
Returns a synchronization object that can be used to lock out writers. Any successful call to Reader.Wait() will stop any attempt at Writer.Wait() until Reader.Release() is called.

IReaderWriterLock.Writer( )

Purpose:
Returns a synchronization object that can be used to lock out any other writer and all readers. Any successful call to Writer.Wait() will stop any attempt at Writer.Wait() and Reader.Wait() until Writer.Release() is called

CriticalProcedure.Init (syncObj)

Purpose:
Calls syncObj.Wait(). syncObj.Release is called by the destructor.

Conclusion

Clarion 6 makes it very easy to take advantage of preemptive threads in your applications. All template generated code uses threaded objects to ensure proper behavior. When you embed code that works with threaded data you don’t have any worries, but when you access shared non-threaded data you should use a synchronization object.