Clarion

Preemptive and Cooperative Thread Models

Who should read this article?

This document is useful for developers looking to migrate their applications from Clarion 5.5 or earlier to Clarion 6, and who wish to understand the new preemptive threads available under Clarion 6. Be sure to read the chapter "**Multi-Threaded Programming**" for a more detailed look at the changes, and information on how to implement thread synchronization.

Threads and Processes

In simple terms a process can be defined as an application. More precisely, a process is a single instance of an application. (In some cases, you can launch more than one instance of a given application; each launch spawns a new process.) Each process owns its own register context and virtual address space, which includes code, global data, stack, and heap memory.

A thread is an independent flow of execution within a process. In that sense, you can think of a process as a container for one or more threads. All threads in a process share the same code, global data, stack, and heap memory, but each thread is apportioned its own region in the stack space.

Cooperative Threading

Cooperative threads are the only types of threads available in versions prior to Clarion 6.

Advantages of Cooperative Threads

They are safe and easy to use. You can, generally, access variables and not have to worry about the contents of the variable changing between two lines of code. The following code will never display the message "Boom" under a cooperative threading system.

```
MyVar  LONG  ! No THREAD attribute

LOOP
  MyVar = THREAD()
  IF MyVar ~= THREAD()
    MESSAGE('Boom')
  END
END
```

Disadvantages of Cooperative Threads

Only one thread is ever active. In other words you cannot have a thread that runs independently of all your other threads; for example a thread that is continuously backing up a data file, or running a report in the background.

Accessing COM objects and some Windows API functions require careful coding and the use of LOCKTHREAD and UNLOCKTHREAD.

Preemptive Threads

Preemptive threads run independently of each other. Only one thread at a time has focus, but focus can change from one line of code to the next. The switching between threads is under the control of the OS.

Advantages of Preemptive Threads

Have you ever built an application where users had to wait while the application performed some lengthy calculation or operation? Then you can benefit from a preemptive thread model.

Understanding threads using a modern operating system's multi-threading capabilities properly, is a fundamental step toward creating fast, responsive applications. Clarion 6 makes creating multi-threaded applications easier than ever.

To understand the power of multithreading you need to know something about how the Windows operating system works under the hood. Windows is a preemptive multitasking operating system. The system CPU can do only one thing at a time, but to give the illusion that multiple processes are running simultaneously the operating system splits the CPU time between the various running processes.

The term preemptive means that the operating system determines when each task executes and for how long. Preemptive multitasking prevents one task from taking up all the processor's time.

The operating system allocates small "slices" of CPU time to each of these processes. Because the CPU is very fast, and the time slices can be extremely small, processes appear to run simultaneously. On a multi-processor system things are a bit more complicated but the basic idea is the same—the operating system divides the time of the CPUs among the processes that need it.

Difficulties in Programming Preemptive Threads

First, keeping track of and switching between threads consumes memory resources and CPU time. Each time the CPU switches to another thread, the state of the current thread must be saved (so it can be resumed again later) and the saved state of the new thread must be restored. With too many threads any responsiveness advantages you hoped to gain may be partially nullified by the extra load placed on the system.

Second, programming with multiple threads can be complex. Creating a single extra thread to handle some background calculations is fairly straightforward, but implementing many threads is a demanding task and can be the source of many hard-to-find bugs. A best practices approach to these potential problems is to use multithreading only when it provides a clear advantage, and then to use a few threads as possible.

Third is the question of shared resources. Because they're running in the same process, the threads of a multi-threaded program have access to that process's resources, including global, static, and instance fields. Also, threads may need to share other resources such as communications ports and file handles. You must synchronize the threads in most multi-threaded applications to prevent conflicts when accessing resources, such as deadlocks (when two threads stop as each waits for the other to terminate).

You need to be **very** careful when accessing non-threaded global variables. The following code will generate the message intermittently when using preemptive threads.

```
MyVar  LONG  ! No THREAD attribute

LOOP
  MyVar = THREAD()
  IF MyVar ~= THREAD()
    MESSAGE('Bang')
  END
END
```

You can protect yourself in this situation by using the IMutex interface declared in CWSYNCH.INT to stop other threads executing at the same time (for a short time)

```
MyVar  LONG  ! No THREAD attribute
MyLocker &IMutex
MutexName CSTRING('MyCode')

MyLocker &= NewMutex(MutexName)
LOOP
  MyLocker.Lock()
  MyVar = THREAD()
  IF MyVar ~= THREAD()
    MyLocker.Release()
    MESSAGE('Bang')
  ELSE
    MyLocker.Release()
  END
END
MyLocker.Kill()
```

If you do not protect yourself when accessing global non-threaded variables you will have an application that will appear to work, but will occasionally do something strange that is hard to reproduce with any consistency.

Programming with pre-emptive threads is **complex**! Even code as simple as

```
MyFunc FUNCTION()
MyResponse = RequestCompleted
RETURN

  MyFunc()
  IF MyResponse = RequestCompleted
    !Do Something
  END
```

may not work how you expected with multiple threads active, if MyResponse does not have the THREAD attribute

Conclusion

Under Clarion 6 you can now take advantage of the power and flexibility of preemptive threads. However, doing so requires that you are very careful with accessing global non-threaded variables.

So what code has to change?

- Code that stores the reference to a threaded variable in a non-threaded variable will not work the same in Clarion 6 as it did in previous versions of Clarion. For an example of this kind of code see ABFILE.CLW in versions of Clarion prior to Clarion 6 where the FileManager (a non-threaded class in standard applications) stores references to its file's record buffer and fields. The file is normally threaded, so the code needed a small adjustment to work in Clarion 6. Hence ABFILE.CLW and the initialization of standard applications have been changed to reflect this fact.

- In versions of Clarion previous to Clarion 6 you did not put the THREAD attribute on a variable that is declared with the EXTERNAL attribute. In Clarion 6 you must put the THREAD attribute on a variable declared with the EXTERNAL attribute when the variable is threaded. Not doing this will cause your application to crash whenever it tries to access the variable from anywhere other than the DLL where the variable is declared.

- In Clarion 5.5 and prior CW versions the runtime library (RTL) had control on thread switching. If the current event was targeted to a WINDOW belonging to a thread other than the previous event was, it generated EVENT:Suspend for previous active thread, EVENT:Resume for the new active thread and then executed the event. This could be any event. In Clarion 6 switching of threads is not within the RTL's control, so EVENT:Resume and EVENT:Suspend events no longer make sense.

Be sure to read the "**Multi-Threaded Programming**" document for a more detailed look at the changes, and information on how to implement thread synchronization.