Multi-Threading in Xbase++

Multi-threading is a concept that is new to Foxpro and Clipper developers, yet it is one of the most powerful features of Xbase++. This seminar introduces new Xbase++ programmers to the concept of multi-threading.

"Multi-tasking" is the capability of the operating system to run several application programs simultaneously. Each application program is embedded in a process. A process contains at least one "thread" where the program code is executed. Program code is executed only in a thread, not in a process. Several threads can be active within a process. This multi-threading capability of the operating system allows various parts of the same application program to run at the same time.

Multi-threading makes it possible to run many procedures and/or GUI dialogs simultaneously. With multi-threading, database work spaces are protected from other threads, i.e. they are "thread safe".

This session introduces the new Xbase++ programmer to the following concepts:

* Why and when to use multi-threading

* How to use multi-threading

* The Thread() object and the Signal() object

* User-Defined Thread() classes

* How to send a message to another thread

* How to synchronize threads

* How to view or dump workspace info in all running threads

* How to view stack info in all running threads

* Programming samples and techniques

* How to create functions and procedures that are "thread-safe".

* How to use threads to protect your database work space.

* The SetTimerEvent() and Sleep() functions.

Multi-Threading in Xbase++

Geek Gatherings Roger Donnay

Why and when to use Multi-Threading

Multi-threading can solve a multitude of problems and also enhance the power of existing applications. The biggest bang for the buck is the ability to run multiple windows simultaneously, even the same window. For example, let's say that your existing application has a "customer" window that provides for browsing and editing customers. With multi-threading, the same procedure can be called multiple times and more than one window can be opened on the desktop.

Here is a code example:

```
FUNCTION Main()

LOCAL oThread, lThread1Active := .f., lThread2Active := .f.

oThread := Thread():new()
Sleep(5)
oThread:start({||ThreadLoop(@lThread1Active)})

oThread := Thread():new()
Sleep(5)
oThread:start({||ThreadLoop(@lThread2Active)})

DO WHILE lThread1Active .OR. lThread2Active
  Sleep(1)
ENDDO

WAIT

RETURN nil

* ---------

STATIC FUNCTION ThreadLoop( lActive )

LOCAL i

lActive := .t.

FOR i := 1 TO 5
  ? 'I am running in thread ' + Alltrim(Str(ThreadId()))
  Sleep(50)
NEXT

USE xtest
XTEST->(dbGoTo(ThreadID()*2))

i := 0
DO WHILE i++ < 5 .AND. !XTEST->(Eof())
  ? 'I am record number ' + Alltrim(Str(XTEST->(RecNo()))) + ' in thread ' +
;
    Alltrim(Str(ThreadId()))
  XTEST->(dbSkip())
  Sleep(50)
ENDDO
```

```
XTest->(dbCloseArea())
lActive := .f.

RETURN nil
```

Each thread also contains a workspace in which up to 65,000 database aliases may be opened. The workspace in a thread is not affected by database operations in other threads, making the workspace "thread-safe".

Look at the above code. Note how the same database can be opened in 2 different threads and be accessing different records. This is a very simple demonstration of multi-threading but adequately explains the concept.

Multi-threading should be used if the application needs to support any of the following tasks:

* Need to open multiple windows with workspace protection.

* Need to run code at a specified time.

* Need a thread for displaying tooltips and balloon tips.

* Need to run multiple tasks simultaneously.

* Need a host server that handles simultaneous connections.

How to use Multi-threading

Multi-threading is one of the simplest concepts to employ in an Xbase++ application. Basically, the Thread() class is used to create a new thread and to run code in that thread.

Example:

```
oThread := Thread():new()
oThread:start("MyFunction")
```

The above example is very simple, but it works very well.

WARNING: If a thread opens files, then the programmer should insure that all files are closed before the thread is terminated, otherwise file handles will remain open by the operating system and there will be no way of closing them until the application is terminated. The simplest way to insure this is to use the :atEnd iVar of the thread object to post a codeblock.

Example:

```
oThread := Thread():new()
oThread:atEnd := {||dbCloseAll()}
oThread:start("MyFunction")
```

The Thread() Object

Every Xbase++ application starts in Thread 1. Each thread is associated with an object of the Thread() class. The function ThreadObject() returns a pointer to the Thread() object for the current thread.

The Thread class provides the Xbase++ programmer with a powerful tool for use in accessing the multi-threading capabilities of the operating system and for allowing an application program to run in multiple threads. Within various threads, different parts of an application run at the same time. An Xbase++ application is always executed in several threads. However, the actual program code written and compiled using Xbase++ normally runs in only one thread while other threads are used for internal purposes, like garbage collection, for example. Using Thread objects, additional threads for program code can be created so that an Xbase++ application can be divided up into different program components that execute independently.

Note: Programming using Thread objects, and creating threads in general, requires careful encapsulation of the program components which are to run in separate threads. Programmers that have never written programs for multi-threading should read the chapter "Multi-tasking and Multi-threading" in the Xbase++ documentation before using thread objects.

iVar	Data Type	Description	ATTRIBUTE
:active	Logical	Indicates whether program code is being executed in a thread.	READONLY
:atEnd	Code Block	Optionally specifies program code to execute once at the end of a thread.	EXPORTED
:atStart	Code Block	Optionally specifies program code to execute once at the beginning of a thread.	EXPORTED
:cargo	Any	Instance variable for ad-hoc use.	EXPORTED
:deltaTime	Numeric	Elapsed time since the last restart of a thread.	READONLY
:interval	Numeric	Time interval for repeated execution of program code in a thread.	READONLY
:name	Character	The name of the Thread	READONLY
:priority	Numeric	Identifies the priority of a thread.	READONLY

```
:result     Any         RETURN value of a thread.                    EXPORTED

:startCount Numeric     Indicates the number of times the execution  READOLY
                        of program code has started in a thread.

:startTime  Numeric/nil Time when a thread starts executing program  READONLY
                        code.

:threadID   Numeric     Numeric ID of a thread.                      READONLY
```

Methods

```
:atEnd( [<xParamList,...>] ) --> NIL

         Reserved method for use by subclasses of the Thread class.

:atStart( [<xParamList,...>] ) --> NIL

         Reserved method for use by subclasses of the Thread class.

:execute( [<xParamList,...>] )

         Reserved method for use by subclasses of the Thread class.

:quit( [<xResult>], [<nRestart>] ) --> NIL

         Terminate the thread of this thread object

:setInterval( <nHSeconds>|NIL ) --> lSuccess

         Sets the time interval for restarting a thread.

:setPriority( <nPriority> ) --> lSuccess

         Sets the priority of a thread.

:setStartTime( <nSeconds> ) --> lSuccess

         Defines the start time for a thread to execute program code.

:start() --> lSuccess

         Executes program code in a thread.

:synchronize( <nTimeOut> ) --> lSuccess

         Synchronizes a second thread with the current thread.
```

The Signal() Object

Signal objects are used to control program flow within one or more threads from one other thread. To achieve this, it is necessary that one and the same Signal object is

visible in at least two threads. Therefore, Signal objects can only be used in conjunction with Thread objects. In other words, at least one additional Thread object must exist in an application for the Signal class to become useful (note: the Main procedure is executed by an implicit Thread object).

There are two methods in the Signal class: :signal() and :wait() . The method :signal() triggers a signal, while the :wait() method causes a thread to enter a wait state. If a thread executes the :wait() method, it waits for the signal and suspends program execution. The thread resumes as soon as a second thread executes the :signal() method with the same Signal object.

With the aid of Signal objects it becomes possible for the current thread to control program flow in other threads. Each thread which executes the :wait() method enters a wait state until another thread executes the :signal() method.

Caution: Multiple Signal objects can be used to control program execution in multiple threads. Since a Signal object limits program control in threads to a wait state, so-called "dead lock" situations can occur when multiple Signal objects are used. A dead lock is reached when thread A waits for a signal from thread B, while, at the same time, thread B waits for a signal from thread A.

```
Instance variables

:cargo    Instance variable for ad-hoc use.   EXPORTED


Methods

:signal() --> self - Triggers a signal for other threads.

:wait( [<nTimeOut>], [<lReset>] ) --> lSignalled

Waits until another thread triggers the signal
```

EXAMPLE3.PRG shows an example where 50 threads are started, all running the same function, yet each one must write to the screen in sequential order, starting from the first to the last. 50 signal objects are used to signal the next thread that the previous thread is finished and now it is it's turn. Run EXAMPLE3.EXE to see the amazing performance of 50 threads communicating in tandem.

User-Defined Thread Classes

The Thread class can serve as superclass for user-defined Thread classes whose instances each have their own thread. Three methods are provided for use in derived

Thread classes. They have the PROTECTED: visibility attribute and can therefore be used in subclasses only. These methods are :atStart() , :execute() and :atEnd() , of which at least the :execute() method must be programmed in a user-defined Thread class. It contains the code to be executed in the separate thread after the :start() method is called.

Here is example code for dumping a database to a comma-delimited file (CSV). Run EXAMPLE6.EXE to see this working.

```
FUNCTION Main()

LOCAL oThread

oThread := DataList():new()
oThread:dataBase := 'CUSTOMER.DBF'
oThread:csvFile := 'CUSTOMER.CSV'
oThread:forCondition := {||.t.}
oThread:start()

DO WHILE oThread:active
  Sleep(10)
ENDDO

? 'Done!', 'Data is in ' + oThread:csvFile

wait

RETURN nil

* ----------

CLASS DataList FROM Thread

EXPORTED:
  VAR database, csvfile, forCondition
PROTECTED:
   METHOD atStart, atEnd, execute
   VAR csvFileHandle, aStruct
ENDCLASS

// Open database file and csv file when thread starts
METHOD DataList:atStart

LOCAL i

IF Empty(::database) .OR. Empty(::csvFile)
  ::quit()
ELSE
  USE (::database) VIA 'FOXCDX'
  ::aStruct := dbStruct()
  ::csvFileHandle := FCreate(::csvFile)
  ::setInterval( 0 )
  FOR i := 1 TO Len(::aStruct)
```

```
    FWrite(::csvFileHandle,::aStruct[i,1] +
IIF(i<Len(::aStruct),',',''))
  NEXT
  FWrite(::csvFileHandle,Chr(13)+Chr(10))
ENDIF

RETURN self

// Dump a record to the CSV file

METHOD DataList:execute()

LOCAL i, cFieldName, xFieldValue, cLine := ''

IF Empty(::forCondition) .OR. Eval(::forCondition )
  FOR i := 1 TO Len(::aStruct)
    cFieldName := ::aStruct[i,1]
    xFieldValue := &(cFieldName)
    IF Valtype(xFieldValue) $ 'CM'
      cLine += '"' + Trim(xFieldValue) + '"'
    ELSEIF Valtype(xFieldValue) = 'L'
      cLine += IIF(xFieldValue,'Y','N')
    ELSEIF Valtype(xFieldValue) = 'D'
      cLine += Transform(xFieldValue,'mm/dd/yyyy')
    ENDIF
    IF i < Len(::aStruct)
      cLine += ','
    ENDIF
  NEXT
  FWrite(::csvFileHandle,cLine + Chr(13)+Chr(10))
ENDIF
dbSkip()
IF Eof()
  ::setInterval( NIL )
ENDIF
RETURN self

// Close database and text files before thread terminates
METHOD DataList:atEnd

dbCloseArea()
FClose(::csvFileHandle)

RETURN self
```

How to view or dump workspace and stack info in all running threads

The ThreadInfo() function returns a 2-dimensional array of information about each running thread. This includes the Thread ID, the function name and line number currently being executed, and the thread object representing the thread.

Unfortunately, this does not give any more information about what is happening in each thread. Often times, it is necessary to know which databases are open in each

thread and the status of those databases, such as which index is being used and what is the record pointer for each database. Also, it can be very helpful to dump the call stack for each thread. This information is usually helpful when an error occurs. The default Xbase++ error handler, ERRORSYS.PRG, only dumps information about the thread in which the error occurred. Most of the time, this is sufficient, but sometimes an error can occur due to conditions in another thread.

To accomplish this, each thread should have an event loop running that is ready to process events and each thread should also have a window object that can receive events. The error handler can then post an event into each thread which will in turn evaluate a code block in that thread. This is done via the PostAppEvent() function. The event loop in each thread would look something like this:

```
FUNCTION EventLoop()

LOCAL nEvent := 0, mp1, mp2, oXbp

DO WHILE nEvent # xbeP_Close
  nEvent := AppEvent(@mp1,@mp2,@oXbp,1)
  IF nEvent == DUMP_THREAD_EVENT
    Eval( mp1 )
  ELSEIF nEvent > 0
    oXbp:handleEvent(nEvent,mp1,mp2)
  ENDIF
ENDDO

RETURN nil
```

The function that is used to initiate the thread dump looks like this:

```
FUNCTION DumpThreads( cDumpText )

LOCAL i, oThread, aThreadInfo := ThreadInfo(THREADINFO_TOBJ)

DEFAULT cDumpText TO ''

FOR i := 1 TO Len(aThreadInfo)
  oThread := aThreadInfo[i,1]
  IF oThread:isDerivedFrom('ThreadDump')
    PostAppEvent( DUMP_THREAD_EVENT,{||DumpThreadInfo(@cDumpText)},,
oThread:window )
  ENDIF
NEXT

Sleep(50)

RETURN cDumpText
```

Each window in which it is desired to dump thread info must be started in a thread that has a custom Thread() object that looks like this:

```
CLASS ThreadDump FROM Thread
```

```
EXPORTED:

VAR window

INLINE SYNC METHOD DumpInfo( cTextDump )

// ThreadData() is a custom function that dumps all required info
// such as work area status, call stack, etc.

cTextDump += ThreadData()

RETURN nil

ENDCLASS
```

Run the DumpThreads.Exe sample program to see how this works. Click on "Dump Threads" to see all thread info in a window. Click on "Crash Program" to see all thread info in an error log.

How to create functions and procedures that are "thread safe"

Care must be taken to insure that variables in multiple threads do not step on each other. The simplest way to do this is to declare variables as LOCAL or PRIVATE. LOCAL and PRIVATE variables are automatically thread-safe because they are not visible to other threads. PUBLIC and STATIC variables, on the other hand, are always visible to other threads.

Converting Non-Gui legacy apps to Xbase++ multi-threading

If you are converting a Clipper or FoxPro 2.6 application to Xbase++ and you wish to take advantage of multi-threading to allow multiple windows, then there are a variety of strategies.

1. Multiple windows that are Non-Gui

This is possible but can be a bit complicated if windows share static data. For example, the @..SAY..GET..READ system utilizes a STATIC object in GETSYS.PRG, therefore data entry in one window can affect the data in another window when using ReadModal(). I overcame this problem by writing a custom ReadModal() that is based on the GetSys.Prg from Clipper and modifying it to be threadsafe. The source is in DCGETSYS.PRG. I have not been successful at
modifying the Xbase++ version of ReadModal().

The other big issue with Non-Gui windows is the issue of SetAppWindow(). Legacy code that uses functions and commands like @..SAY, @..PROMPT, SetColor(), SaveScreen(), etc. will create a runtime error if the SetAppWindow() function does not point to an XbpCrt() class object. The XbpCrt() class was not intended to be used in multi-threaded applications, however I have been successful at helping Clipper developers to convert legacy code to use multiple XbpCrt() windows running in separate threads. The first obstacle is to insure that when an XbpCrt() window receives focus that it also becomes the SetAppWindow(). This is accomplished like so:

```
oCrtWindow:setInputFocus := {|a,b,o|SetAppWindow(o)}
oCrtWindow:setDisplayFocus := {|a,b,o|SetAppWindow(o)}
oCrtWindow:close := {|a,b,o|_Keyboard(K_ESC)}
```

Run the NONGUI.EXE sample program to see how this works.

2. Privatizing Public variables

Many legacy applications use a lot of PUBLIC variables. Converting such applications to multi-threading can often be daunting because programmers are often told to use only LOCAL or STATIC variables in state-of-the-art applications. In my experience, the effort to remove all PUBLICS and PRIVATES from legacy apps usually leads to failure because it almost always requires a major redesign of the program architecture and will consume much more time
than originally planned.

Look at the below example. Without the PRIVATE declarations, the ReadTest() function would overwrite PUBLIC variables. This technique allows PUBLICS to be inherited as PRIVATE even though the private variables have the same name. Run the NONGUI.EXE sample to see this effect.

```
FUNCTION Main()

PUBLIC cString := Space(20)
PUBLIC nNumber := 1234.56
PUBLIC dDate := Date()

FOR i := 1 TO 3
  CrtRun( {||ReadTest() } // Open CRT window in new thread
NEXT

WAIT

RETURN nil

* --------
```

```
FUNCTION ReadTest()

LOCAL GetList[0]

PRIVATE cString := M->cString
PRIVATE nNumber := M->nNumber
PRIVATE dDate := M->dDate

@ 1,10 SAY 'Enter a String' GET M->cString
@ 2,10 SAY 'Enter a Number' GET M->nNumber PICTURE '9999.99'
@ 3,10 SAY 'Enter a Date  ' GET M->dDate

ReadModal(GetList)

RETURN nil
```

3. Making STATIC Variables Thread-Safe

Legacy programmers can tend to use STATIC variables within a source file to pass data between functions rather than passing them as LOCALs or PRIVATEs. This is never a good idea, however it could take more effort and cause more regression to rewrite all the code than to do a simple workaround. If a procedure is being called concurrently in different threads, then STATIC data from one thread will be overwritten by another thread.

Look at the below example. Obviously this is not a good coding method and it would be best to pass the date down to SetTheDate(), and if this were a small program, that would be my suggestion. On the other hand, I have seen apps where there are many STATICS being used by many sub-functions, and therefore a workaround may be a better solution. The workaround assumes that there will never be more than 20 threads running in the application, so a 20-element static array is created, and then a #define is used at the top of the source file to negate the need to change the names of all the variables throughout the source.

```
// Before Workaround

STATIC sdDate

FUNCTION MyApp

scDate := Date()
SetTheDate()

// Do more stuff

RETURN sdDate

* ---------

FUNCTION SetTheDate()
```

```
@ 1,10 SAY 'Enter Date' GET sdDate

READ

RETURN nil


// After Workaround

STATIC saDate[20]
#define scDate saDate[ThreadID()]

FUNCTION MyApp

scDate := Date()
SetTheDate()

// Do more stuff

RETURN sdDate

* ------------
```

4. Making Object Methods Thread-Safe

The SYNC METHOD of a class is used to insure that multiple threads running instance methods of the same class do not occur simultaneously. Each thread will wait until the called method is completed before allowing another thread to run that same method.

Examples of SYNC METHOD are included in the COFFEE.PRG example and also the THREADDUMP.PRG example.

How to send a message to another thread

Many times it is necessary to control the behavior of another thread that is running code. This can be done in a variety of ways.

1. Pass variables by reference to the other thread.

In the below sample, two variables are passed from the main thread to a function in a new thread. The values of these variables are changed in the main thread and are visible in the new thread. Run EXAMPLE4.EXE to see this behavior.

```
FUNCTION Main()

LOCAL oThread, lIsRunning := .f., cTime := Time(), i

oThread := Thread():new()
```

```
oThread:start({||TestLoop(@lIsRunning,@cTime)})

FOR i := 1 TO 10
  cTime := Time()
  Sleep(100)
NEXT

lIsRunning := .f.

WAIT

RETURN nil

* ----------

FUNCTION TestLoop( lIsRunning, cTime )

LOCAL i

lIsRunning := .t.

FOR i := 1 TO 1000
  ? i, cTime
  Sleep(10)
  IF !lIsRunning
    EXIT
  ENDIF
NEXT

RETURN nil
```

2. Post an EVENT to another thread.

The DumpThreads() function discussed earlier in this session shows an example of posting an event to another thread that has a window and an event loop running.

3. Use a Signal() object to control another thread

In the below example, the 2nd thread waits for a signal from the first thread before continuing. The cargo of the Signal() object is also used to pass information to the 2nd thread. Run EXAMPLE5.EXE to see this behavior.

```
FUNCTION Main()

LOCAL oThread, oSignal, i

oThread := Thread():new()
oSignal := Signal():new()
oSignal:cargo := {.t.,Time()}
oThread:start({||TestLoop(oSignal)})
```

```
FOR i := 1 TO 10
  oSignal:cargo[2] := Time()
  oSignal:signal()
  Sleep(100)
NEXT

oSignal:cargo[1] := .f.

WAIT

RETURN nil

* ----------

FUNCTION TestLoop( oSignal )

oSignal:wait()
DO WHILE oSignal:cargo[1]
  ? oSignal:cargo[2]
  oSignal:wait()
ENDDO

RETURN nil
```

How to Synchronize Threads

Alaska Software provided a sample of advanced multi-threading in their COFFEE.PRG program.

This example is a real life simulation of a team of programmers and a single coffee machine. It shows how to solve special problems that may arise in multi-threaded programs.

The program demonstrates two programming techniques for multi-threading using a classical problem: multiple threads access and change shared program resources at the same time and must be coordinated. Thread coordination is accomplished using one SYNC method (serialize program execution in multiple threads) and two Signal objects (halt/restart program flow in one thread).

Run the COFFEE.EXE sample to see this behavior.

The SetTimerEvent() function

```
SetTimerEvent( [<nNewInterval>], [<bNewBlock>]   )  -->
   { nOldInterval, bOldBlock }
```

<nInterval> is a numeric value indicating the time interval in 1/100 seconds at which the code block <bBlock> is executed. If the value zero is passed, the code block is no longer executed.

<bBlock> is a code block automatically executed in a separate thread at specific time intervals.

The function SetTimerEvent() starts a thread independent of the running program in which the code block <bBlock> is evaluated at a consistent time interval. In this separate thread, other components of the program can be executed but they must be self contained and independent of the main program. A common example for the timer thread is the display of the time (see example). But more complex program components, like print jobs, can be set to run without the main program being disturbed.

Warning: Output on the screen from the timer thread should not be performed with QOut(), DispOut() or DevOut() without first saving the cursor position and then restoring it. Otherwise, the cursor position changes in the main program. When screen output must occur in the timer thread, it is preferable to use the function DispOutAt() which does not change the cursor position.

In the below example, the time is displayed once per second. The timer thread continues running even though the main program is receiving input from the user using READ.

```
PROCEDURE Main

LOCAL cVar1:= "Xbase++" , cVar2:="for 32bit"

SetTimerEvent(100, {|| DispOutAt(0, 0, Time()) } )

@ 10, 0 Say "Variable1" GET cVar1
@ 12, 0 Say "Variable2" GET cVar2
READ

SetTimerEvent(0)

RETURN
```

The Sleep() Function

The function Sleep() defines an explicit idle condition for the thread in which the function is called. During the idle condition the thread uses no system sources (CPU time) and leaves resources available to other threads. The idle condition lasts <nTime> hundredths of a second and cannot be interrupted.

Idle conditions can also be defined with the function InKey() or AppEvent(). But with these functions a program waits until either an event takes place or until the defined time interval has passed. Neither function guarantees that the idle time interval will be completed. In addition, events can be lost when an idle condition is defined with Inkey() or AppEvent(). With Sleep() the time interval for the idle condition is guaranteed. Events which occur during the idle condition are stored in the event queue and are available after the completion of Sleep().

```
Sleep( <nTime> ) --> NIL
```

<nTime> is a positive integer which specifies the time interval in 1/100 seconds during which a thread is to remain in an idle condition.

The below example attempts to lock a data record. If a lock cannot be set, the program waits 1/2 second and attempts to lock the record again. After five attempts, termination occurs with a warning notice.

```
PROCEDURE Main

LOCAL nMaxRetry := 5, nTries := 0, lUpdated := .F.

USE Customer

DO WHILE nTries < nMaxRetry
  IF ! RLock()                        // attempt to lock record
    Sleep( 50 )                       // wait 1/2 second
    nTries ++                         // if lock fails
  ELSE
    REPLACE Customer->Name WITH "Jones" // update data record
    DbUnLock()
    lUpdated := .T.
  ENDIF
ENDDO
                                      // data record is locked
IF ! lUpdated                         // more than 2.5 seconds
  Alert( "Data record is currently locked", {"Ok"} )
ENDIF

USE

RETURN
```

Conclusion

```
oThread:atEnd := ;
    {||MsgBox('Thank you for attending this session!')}
```