Module13: APL Threads

Windows divides its workload into tasks or *processes*. Each process is allocated virtual address space and given control of some resources. A *thread* is the smallest kernel-level object of execution. When a process is created, a primary thread is generated along with it. This thread is then scheduled to run on a processor. After the primary thread has started, it can create other threads that share its address space and system resources but have independent contexts. Threads, like processes, can time-slice a processor's throughput leading to the illusion of parallel processing on single-processor machines, which are, usually, most of the time looping idly.

Dyalog APL runs in a ***C thread***. When APL starts up it generates an ***APL thread***, called the *base thread* or root thread, which can create other APL threads, each with their own execution stack and state indicator. Thus Dyalog supports parallel processing of APL code via multi-threading whereby more than one APL expression can apparently run concurrently. This allows background calculations to run at the same time as interactive tasks, which greatly improves system responsiveness from a user's point of view.

§ 13.1 Spawning a new Thread

§§ 13.1.1 The Spawn Operator, &

A thread is initiated by an asynchronous call on a monadic or dyadic function using the new monistic primitive operator *spawn* (&).

```
      {TID}←{⍺}f&⍵                    ⍝ Runs function f in a new thread with ID TID
```

The (shy) result of the derived function `f&` is the identity of the thread in which `f` is being run. When the (ambivalent) function `f` terminates, its result (if any) is, by default, returned in the session.

```
      t←3×&4
12

      t
1
```

The result returned by the derived function `×&` is **not** the result of the multiplication, but is the unique thread number of the newly created thread – in this case `⊢t←1`. We denote this behaviour by

```
      3×&4
2
12
```

The thread number is now 2, the next available positive integer.

An analogous situation arises in the case of executing a diamondized statement whereby the result of an expression is not necessarily that which is displayed in session.

```
      r←⍎'33⋄44⋄55'
55
33
44
```

Compare with

```
      r←⍎&'33⋄44⋄55'
3
33
44
55
```

which is run in thread number 3, or

```
      +⍤⎕dl&5
4
5.078
```

which is run in thread number 4. Thread ID's are allocated sequentially from 0, the base thread ID, to `¯1+2*31 ↳ 2147483647`, at which point, the sequence 'wraps around' and numbers are allocated from 1 again, avoiding any still in use. The counter may be reset to 0 by `)RESET`.

Functions that take a significant length of time to return their result may be run in the background if their results are not immediately required.

```
      #.⍎&'S←OLEServers'
```

Niladic functions can be accommodated by way of execute, or with a monadic dfn:

```
      ⍎&'Niladic'      ⍝ because Niladic& is syntactically incorrect
      {Niladic}&0      ⍝ argument 0 is discarded by monadic dfn
```

13.1.1.1 What would happen if you run function `{+∇&⍵}` on any argument?

A thread can spawn any number of new sub-threads. This implies a hierarchy of parent and child threads whose ancestral root is ultimately in base thread number 0. Children of a terminated parent thread are adopted by the grandparent.

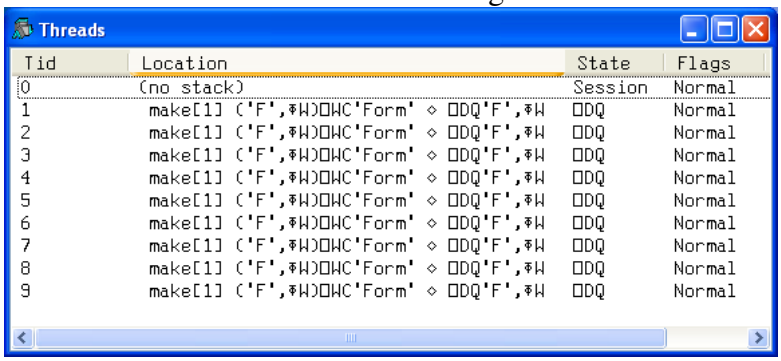
Many parallel threads can be initiated by using each (`¨`) in conjuction with spawn (`&`) because `f&¨` is equivalent to `(f&)¨`. Compare this with `f¨&` which is equivalent to `(f¨)&` which launches only one new thread.

13.1.1.2 Use the function `∇make∇` to initiate a number of `⎕DQ`'ed `Form`s in parallel.

```
      ∇ make W
[1]     ('F',⍕W)⎕WC'Form' ⋄ ⎕DQ'F',⍕W
      ∇
```

In order to monitor and debug applications involving many threads a new *threads tool* has been introduced in Dyalog version 10.1 (see Dyalog APL Version 10.1 Release Notes). The new tool may be opened from the session menu by [Threads][Show Threads…] or pop-up menu [Threads…]. It too has a pop-up menu.

13.1.1.3 Open the threads tool and view the threads created using `∇make∇` above. .



Delete each Form and observe the corresponding thread disappear.

13.1.1.4 Examine the function `∇thrd∇` which recurs until the SI stack is 4 levels deep.

```
      ∇ thrd w  ⍝ start with thrd 0 after )RESET
[1]     :If (¯1+⍴⎕SI)<3
[2]         thrd&w,1 ⍝ previous,
[3]         thrd&w,2
[4]     :End
[5]     ⎕DL ?10 ⍝ threads are adopted when parent disappears
      ∇
```

Run the expression

```
thrd 0
```

and watch the threads disappear in the thread tool.

§§ 13.1.2 Thread Identity from ⎕TID and ⎕TNAME

Each thread has a positive integer ID. The ID of the current thread may be found by means of a system command,

```
)TID                                    ⍝ Reports identity of current thread
```

or by way of a system function inside an APL program,

```
TID←⎕TID                                ⍝ Current thread number
```

The result `TID` is a simple positive integer scalar, dataType ZSc. The base thread, which is always present, has `⊨⎕TID≡0`. This identity assumes that the base thread is the current thread. Otherwise

```
)RESET
      ⍎&'⎕TID' ↳ 1
1
```

```
      ⍎&'⎕TID' ↳ 2
2
```

Each thread can also be given an arbitrary name, dataType CVec, using the new system variable,

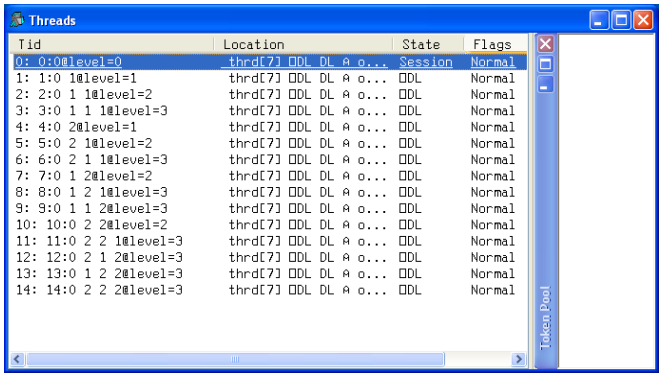
```
⎕TNAME                                  ⍝ The name of the current thread
```

Initially, in a new thread `⊨⎕TNAME≡''`.

13.1.2.1 Modify the function `∇thrd∇` as below and create a global variable `DL` with a suitable delay value, say 10 seconds

```
      ∇ thrd w  ⍝ start with "thrd 0" after )RESET
[1]     ⎕←⎕TNAME←(⍕⎕TID),':',(⍕w),'@level=',⍕¯1+⍴⎕SI
[2]     :If (¯1+⍴⎕SI)<3
[3]         thrd&w,1 ⍝ previous,
[4]         thrd&w,2
[5]     :End
[6]     ⎕DL DL
      ∇
```

Open the Threads tools, reset the SI stack and trace `thrd 0` to line `[5]`. Right click on one of the threads in the Threads tool and select [Auto Refresh] and [Switch to]. See new trace window with new current thread in the caption. A star appears against this thread in the Threads tool indicating that it is suspended. Use `)TID` to verify that this is now the current thread. View the `)SI` stack. Reset and save the workspace.



When more than one thread is running, the `)SI` stack is a branching tree originating from the root (base) thread. If a thread sustains an untrapped error then execution of the thread is suspended and all other threads are paused. The session is attached to the suspended thread making is possible to examine local

variables and trace through the code. Error messages are prefixed with thread numbers. More information on debugging threads can be found in the Dyalog version 10.1 *Release Notes*. In particular, the session supports a number of new facilities for examining thread states.

Threads are flagged in the Threads tool as either normal or paused. A *paused thread* is one that has temporarily been removed from the list of threads that are being scheduled by the thread scheduler. A paused thread is effectively frozen. Runaway threads may be paused with the [Pause All] item in the Thread Tool pop up menu.

It is possible to switch suspension to a different thread, but not to a pendent thread, using the system command `)TID` with a thread ID parameter.

```
)TID TID                          ⍝ Switch to suspension to thread number TID
```

This suspends a running thread and opens a new trace window on the thread, making it the current thread.

§§ 13.1.3 Thread Numbers with ⎕TNUMS and ⎕TCNUMS

`⎕TNUMS` returns all the thread numbers corresponding to initialised threads.

```
TIDs←⎕TNUMS                       ⍝ The numbers of all threads
```

The result of the niladic system function is a positive integer vector, dataType ZVec. `⊢0∈⎕TNUMS`

Each thread may have child threads. The resulting hierarchy may be analysed using the system function `⎕TCNUMS` that reports only the child threads of the argument threads.

```
ChildTIDs←⎕TCNUMS ParentTIDs ⍝ The numbers of all child threads of given parents
```

`ParentTIDs` is a simple array of thread numbers (dataType ZArr), and `ChildTIDs` is a simple vector of thread numbers (dataType ZVec), or zilde if there are none.

13.1.3.1 In the function `thrd` above, set `DL` to `60` and trace into (**Ctrl+Enter**) `thrd 0`. Hit **Enter** until you reach line [5]. This will initiate 14 new threads and keep them alive for a minute. Explore the results of `⎕TNUMS` and `⎕TCNUMS`.

It is possible to terminate threads, and optionally (and by default) their dependents, under program control with system function `⎕TKILL`:

```
{Terminated}←{Descendents}⎕TKILL TIDs   ⍝ Terminate threads/families in TIDs
```

The Rarg is a simple array of thread IDs (ZArr), Larg is a Boolean determining the fate of descendents (default is 1, terminate entire progeny). The result is a simple vector of actual terminations (ZVec).

```
⊢⍬≡⎕TKILL 0 ⍝ the base thread is always present.
```

If an intermediate thread is terminated then the thread's parent adopts its children.

13.1.3.2 Add new line `[2]` to the function `∇thrd∇` and replace the delay with an infinite loop.

```
      ∇ thrd w
[1]     ⎕←⎕TNAME←(⍕⎕TID),':',(⍕w),'@level=',⍕¯1+⍴⎕SI
[2]     ⍝⎕←'0',⍕{⎕ML←1 ⋄ ∈⍵}(¯1+⍴⎕SI)⍴⊂'⎕tcnums¨'
[3]     :If (¯1+⍴⎕SI)<3
[4]         thrd&w,1
[5]         thrd&w,2
[6]     :End
[7]     I←0       ⍝ I is local to thread
[8]     :While 1 ⍝ otherwise thread disappears->adoption
```

```
[9]         :If {(⍵÷1000000)≡⌊⍵÷1000000}I ⍝update showTree every 1E6
[10]            updateTree w ⎕TNAME I
[11]        :End
[12]        I←I+1
[13]    :End∇
```

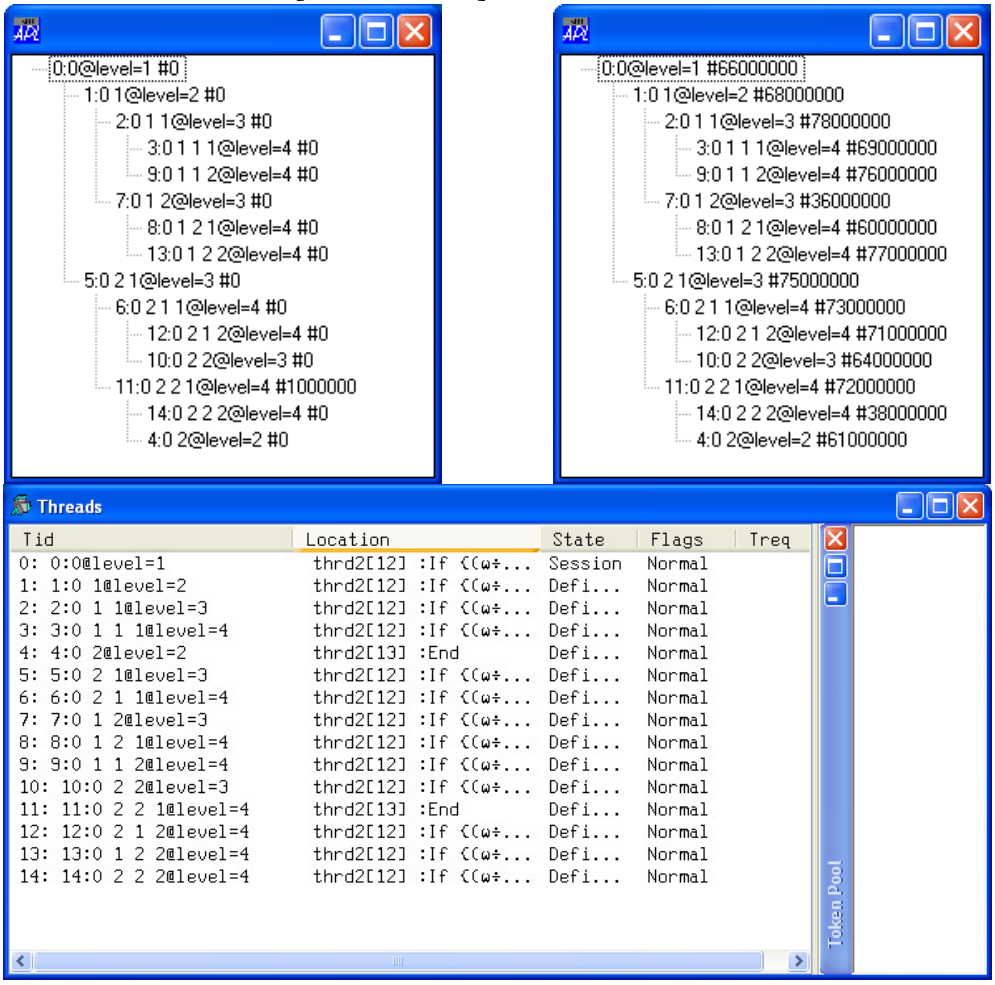
where the function `∇updateTree∇` is

```
      ∇ updateTree(W Name I);IDs;Ind
[1]     IDs←↓⍉3 3 3 3⊤0 9,(11+⍳6),20+⍳6 ⍝ list possible IDs
[2]     Ind←IDs⍳⊂4↑⍵                    ⍝ look for current ID
[3]     IDs←F.TV.Items
[4]     IDs[Ind]←⊂Name,' #',⍕I          ⍝ build item label
[5]     F.TV.Items←IDs
[6]     {F.TV.Expanding ⍵}¨⍳15          ⍝ NB can't do 1 ⎕NQ... in thread
[7]    ∇
```

Run the niladic function `∇showTree∇`. This sets up a `TreeView` of the coming thread hierarchy.

```
      ∇ showTree
[1]     'F'⎕WC'Form'('Posn' 70 75)('Size' 25 20)
[2]     'F.TV'⎕WC'TreeView'('Size' 100 100)
[3]     F.TV.Items←15⍴⊂,'-'
[4]     F.TV.Depth←0 1 2 3 3 2 3 3 1 2 3 3 2 3 3
[5]     F.TV.HasButtons←0
[6]     {F.TV.Expanding ⍵}¨⍳15∇
```

Open the Threads tool and check the [Auto Refresh] item. Run `thrd 0` after a `)RESET`



Watch the tree update every million counts. Notice the update order is not predictable. Hit **Ctrl+Break** in the session. Notice the suspended thread is no longer updated. Break again. View the `)SI` stack. Experiment with [Restart All], [Pause All], [Resume All], `⎕TKILL`, [Strong Interrupt] in the System Tray and [Action][Interrupt] in the Session.

§ 13.2 MultiThread Interactions

§§ 13.2.1 Thread Synchronisation with ⎕TSYNC

Often it is necessary to wait for the result of a thread before another program can run. This situation is managed using ⎕TSYNC, which takes an argument of a simple array of thread numbers. ⎕TSYNC waits until the thread initiating function has finished and all the results, if any, have been produced. It then returns an array of the same outer shape as the argument, each thread result (if there is one) being an enclosed element of the array in the corresponding position.

```
{ArrArr}←⎕TSYNC ZArr      ⍝ Wait for and return results of all threads in ZArr
```

If a thread is subject to an active ⎕TSYNC, the thread result appears as the result of ⎕TSYNC rather than in the session.

If one thread is waiting for another to finish and that one is waiting for another in a cyclic dependency then a trappable DEADLOCK error (number 1008) is generated. This error is also generated if you attempt to wait for the base thread to finish by ⎕TSYNC 0.

§§ 13.2.2 Holding Tokens with :Hold

When many threads wish to access the same resource then a method of synchronising and controlling access is needed. Access is controlled by *tokens*, arbitrary character vectors identifying entry into critical sections of code.

```
:Hold VecCVec ⋄ ... ⋄ :Else ⋄ ... ⋄ :EndHold   ⍝ Attempt to acquire tokens
```

The control structure initiated by :Hold blocks entry into the next segment of code until all the tokens in the vector of character vectors (VecCVec) have been acquired. If no other :Hold has acquired a token then it may be acquired by the current thread. The token is released on exit from the structure.

If a :Else clause has been included then execution proceeds into the :Else clause if all the tokens in VecCVec are not available. Each token may only be held once in the workspace. Trailing blanks are ignored. Holds may be nested in a cumulative fashion, which gives a further danger of DEADLOCK.

Thus, function below always results in a DEADLOCK. (Note dfns do not support controls structures.)

```
      ∇ foo
[1]     :Hold '⍳∇|' '7896' '#.#.#' ''
[2]         :Hold '⍳∇|'
[3]             ⍝ never get here
[4]         :End
[5]     :End∇
      foo
DEADLOCK
foo[2] :Hold '⍳∇|'
       ^
```

A list of all tokens which have been acquired or requested by a :Hold control structure can be displayed by the system command)HOLDS.

```
)HOLDS                                  ⍝ Reports all tokens acquired or requested by :Holds
```

This command displays all the tokens that have been acquired or requested, one per line. The token is followed by a colon and then the (one and only) thread number which has acquired the token followed by all the threads which are currently requesting it.

For example, given the *DEADLOCK* in force above,

```
      ⍎&'foo' ↳ 1
      ⍎&'foo' ↳ 2
      )HOLDS
:       0       1       2
#.#.#:  0       1       2
7896:   0       1       2
⍳∇|:    0       1       2
```

§§ 13.2.3 Pooling Tokens with ⎕*TPUT* and ⎕*TGET*

Dyalog version 10.1 contains an alternative method for synchronising threads. A pool of tokens is maintained from which tokens may be acquired, when available, and into which tokens may be deposited.

In this context, a *token* has a different meaning from that in section 13.2.2. Tokens are no longer character strings. They are represented by a non-zero integer scalar *type*, and may optionally have an arbitrary array *value*.

The pool may contain up to `2*31 ↳ 2147483648` tokens. They are identified by their type and are managed in a FIFO (first in first out) fashion and therefore do not have to be unique.

You can put a token, identified by its type and the sequential order in which it was deposited, into the pool of tokens.

```
{TIDs}←{Values}⎕TPUT Types      ⍝ Puts token Types in pool, freeing threads TIDs
```

Types (dataType ZVec) is be a vector of token types. *Values* (dataType VecArr) is an optional vector of values associated with each corresponding token. The default value is the type itself. The result, if any, of ⎕*TPUT* is a vector of thread numbers that have been unblocked by the introduction of the new token(s) into the pool.

Let us put two tokens both of type 29 into the pool:

```
      ⍬≡⎕TPUT 29 ↳ 1
      ⍬≡⎕TPUT 29 ↳ 1
```

The niladic system function ⎕*TPOOL* returns the type of every token in the pool.

```
Types←⎕TPOOL                    ⍝ Returns Types of tokens in the pool
```

Types is a simple integer scalar or vector of token types, or zilde if empty. Thus

```
      ⎕TPOOL ↳ 29 29
```

The ambivalent system function, ⎕*TGET*, retrieves tokens from the pool and returns their values. Negative tokens may be retrieved any number of times. Positive tokens are removed from the pool when they are retrieved.

```
{Values}←{TimeOut}⎕TGET Types ⍝ Gets token Types out of FIFO pool, when available
```

Types is a simple integer scalar or vector that specifies one or more tokens. *TimeOut* is a maximum time in seconds to wait for a response (dataType NSc). *Values* is an arbitrary array value in the case of a single token, or a vector of array values in the case of more than one token being retrieved. ⎕*TGET* returns only when the tokens are available (or in the event of a timeout in which case zilde is returned).

13.2.3.1 Try to get token type 29 when the pool is empty.

```
      ⎕TGET& 29  ⍝ try to get token type 29
```

Note the appearance of a new thread in the Threads tool. Now place a token, type 29, in the pool.

```
      ⎕TPUT 29
```

Try again and this time give the token a value.

```
      +∘⎕TGET& 29
      ⎕A ⎕TPUT 29
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The output came from the ⎕TGET and there are now no tokens in the pool. If we had put a negative type in the pool then it could be retrieved any number of times as a positive type without being removed from the pool. But getting a negative type removes it from the pool.

All tokens can be removed from the pool by ⎕TGET⎕TPOOL. Note that ⎕TGET 0 can only be stopped by a [Strong Interrupt] (from the System Tray icon).

13.2.3.2 Experiment with ⎕A⎕TPUT¯29, +∘⎕TGET&29 and +∘⎕TGET&¯29. Also experiment with a request for two such tokens with, for example, +∘⎕TGET&-29 29 and ⎕A⎕AI⎕TPUT¨¯29.

Outstanding token requests from calls to ⎕TGET in various threads can be found from the result of system function ⎕TREQ.

```
      Types←⎕TREQ TIDs                    ⍝ Current token requests for all thread identities TIDs
```

⎕TREQ takes a vector (or scalar) Rarg of thread IDs, and returns a vector of all the requested token Types (ZVec or zilde) in all threads in TIDs (positive ZVec).

13.2.3.3)RESET to clear the pool. Put 26 tokens into the pool, with values 'A', 'B', .. 'Z'. Note the contents of the Token Pool, which can be docked in the Threads tool. In thread 1, get all the values of 8 5 12 12 15 in a single ⎕TGET request. Use ⎕TREQ to examine the outstanding token requests, or look in the Treq column of the Threads tool. Put another token 12 with value 'L' into the pool.

§ 13.3 General Thread Programming

§§ 13.3.1 Thread Switching

If you execute more than one Dyalog APL thread, a maximum of only one thread is actually running at any instant; the others are paused. Each APL thread has its own State Indicator, or SI stack. When APL switches from one thread to another, it saves the current stack (with all its local variables and function calls), restores the new one, and then continues processing.

Execution may switch from one thread to another only at certain critical points in the code. Generally, execution may not switch mid-line. But the interpreter may switch to a different APL thread

at the end of every line of code.

Therefore, one very useful way to ensure that two primitive expressions are executed sequentially without interference from other threads is to place the two expressions on the same line, separated by a **diamond separator** (⋄).

Global names set on one line of a function might not have the same value on the next line if other threads access them.

Local names follow the same name scope rules in threads as they do without threads, except that each thread stack should be viewed as an independent program stack as far as visibility of local names is

concerned. Local names created at a point in the code after the thread has been spawned are visible to the code in that thread thereafter, but are not visible to threads spawned at an earlier stage. (See, for example, the unlocalised variable `I` in the function `∇thrd[7]∇` above.)

If an intermediate thread is terminated, the grandparent adopts the child threads. This can cause names to change scope and so, generally, all variables should be localised inside functions, especially in multi-thread environments.

There are other times at which execution may switch from one thread to another. These are points at which the interpreter is waiting in an idle state for input:

```
⎕DQ ⎕DL ⎕FHOLD ⎕ED ⎕SR ⎕ ⍞ :Hold.
```

Other times when the interpreter might execute code in other threads are: while waiting for input from the **⎕SE session**, while waiting for a **⎕NA function** call to finish, while waiting for an **AP function** to finish or while waiting for an **OLE function** to terminate. Thread interjection can be controlled with `:Hold`.

§§ 13.3.2 External Threads with ⎕NA

Normally a `⎕NA` call runs in the same C-thread as APL itself. In order to make the call run entirely in the background it must be run in a separate C-thread. This can be done by placing an `&` after the name of the function in the `⎕NA` definition and calling the external function, so defined, through the spawn operator.

Consider the four different ways of running the external function, `Sleep`.

1. Calling `Sleep` in the normal way, for, say, 10 seconds (=10,000 milliseconds), causes the APL session to completely cease responding for the duration:

```
      ⎕NA'kernel32|Sleep I4' ⋄ Sleep 10000  ⍝ EVERYTHING FREEZES
```

2. Calling `Sleep` in a separate APL thread the normal way through the spawn operator, for, say, 10 seconds, also causes the APL session to stop responding for the duration and does not continue processing code until the 10 seconds has elapsed:

```
      ⎕NA'kernel32|Sleep I4' ⋄ Sleep&10000  ⍝ EVERYTHING FREEZES
```

3. Calling `Sleep`, having defined it in the `⎕NA` with a trailing `&`, causes the APL session to be partially active but again all further processing of APL code is frozen until the 10 seconds has elapsed:

```
      ⎕NA'kernel32|Sleep& I4' ⋄ Sleep 10000 ⍝ SOME MENUS ACTIVE
```

4. Calling `Sleep` through the spawn operator, having defined it in the `⎕NA` with a trailing `&`, causes the APL session to actively respond and further processing of APL code enabled immediately:

```
      ⎕NA'kernel32|Sleep& I4' ⋄ Sleep&10000 ⍝ ALL SESSION ACTIVE
```

It is not possible to thread the `⎕DQ` function directly.

```
      ⎕DQ&'msg'⎕WC'MsgBox'
DOMAIN ERROR
```

This is because `⎕DQ` can only be run on objects in the same thread. It can, however, be run under cover.

```
      ⎕FX'msg w' '⎕DQ''msg''⎕WC''MsgBox'''
      msg&1
```

13.3.2.1 Repeat this with a `Form` and note that APL is free to continue processing in the parent thread after the `Form` has been `⎕DQ`'ed in a separate thread.

In the special case of a `MsgBox`, and modal dialogue boxes such as a `FileBox`, all processing in other APL threads is suspended until the message box has been dismissed.

13.3.2.2 Define a message box external function `∇mbx1∇` in the normal manner, and a second function `∇mbx2∇` that will run in a separate C thread.

```
'mbx1'⎕NA'I user32|MessageBoxA I <0T <0T I'
'mbx2'⎕NA'I user32|MessageBoxA& I <0T <0T I'
```

Call the functions in the four different ways; for example,

```
mbx1 0 'call unthreaded' 'NA unthreaded' 1
```

and note the different responses. In particular note that this enables the special case of a message box to run in a separate C-thread and free other APL-threads to continue processing.

One significant advantage of multi-threaded DLL calls that are run in separate C thread is the fact that they, unlike APL threads, can take advantage of multiple processors, if the operating system allows it.

Once a C-thread has been started it is maintained in the APL-thread for subsequent use in that thread and is discarded when the APL thread finishes. `⎕NA` calls that are to be run concurrently in separate APL-threads should be 'thread-safe'. Note that standard Windows API functions *are* thread safe.

`⎕NA` calls that interact with Dyalog GUI objects should generally be run in the same C-thread and therefore should not be multi-threaded.

§§ 13.3.3 Threading callback Functions

All GUI objects are owned by the thread that created them. The Root object (`#`) and the Session objects (`⎕SE`) are *owned* by the Base thread (`0`). If a thread is terminated then any objects that it owns become owned by the parent object.

All the events generated by an object are reported to the thread that owns the object and cannot be detected by any other threads. The only exception to this rule is events associated with `TCPSocket` objects. Because of the danger of losing TCP events, which should be processed immediately, events from `TCPSockets` can be seen in every thread.

There is a special syntax associated with threading callback functions of GUI objects. Ampersand (`&`) is simply appended to the name of the callback function when it is associated with the object `Event` property, in a similar fashion to that used in threaded `⎕NA`. This syntax applies equally to niladic callback functions.

13.3.3.1 Consider a `Form` with a niladic callback `∇draw∇` that draws a `Poly` line on each `MouseMove` event. The `Form` may be created by

```
w←'Form'('Coord' 'Pixel')('BCol' 0 0 0)
w,←('Size' 200 200)('Event' 'MouseMove' 'draw')
'F'⎕WC w
```

where the callback function is defined as

```
     ∇ draw;y1;x1;y2;x2        ⍝ line, lenght 10, random angle
[1]    ⎕DL 0.1                 ⍝ waste some time
[2]    y1 x1←?2⍴200            ⍝ random line origin
[3]    x2←(?21)-11             ⍝ ¯10 ≤ x ≤10
[4]    y2←((10*2)-x2*2)*0.5    ⍝ since r²=x²+y² for a circle
[5]    y2 x2+←y1 x1            ⍝ add random origin
[6]    'F.'⎕WC'Poly'('Points'(2 2⍴y1 x1 y2 x2))('FCol'(?3⍴255))∇
```

Using the mouse, drag your sprite over the *Form* and, at the same time, type in the session at the cursor position. Note the response. Now define the *Form* with

```
'F'⎕WS'Event' 'MouseMove' 'draw&'
```

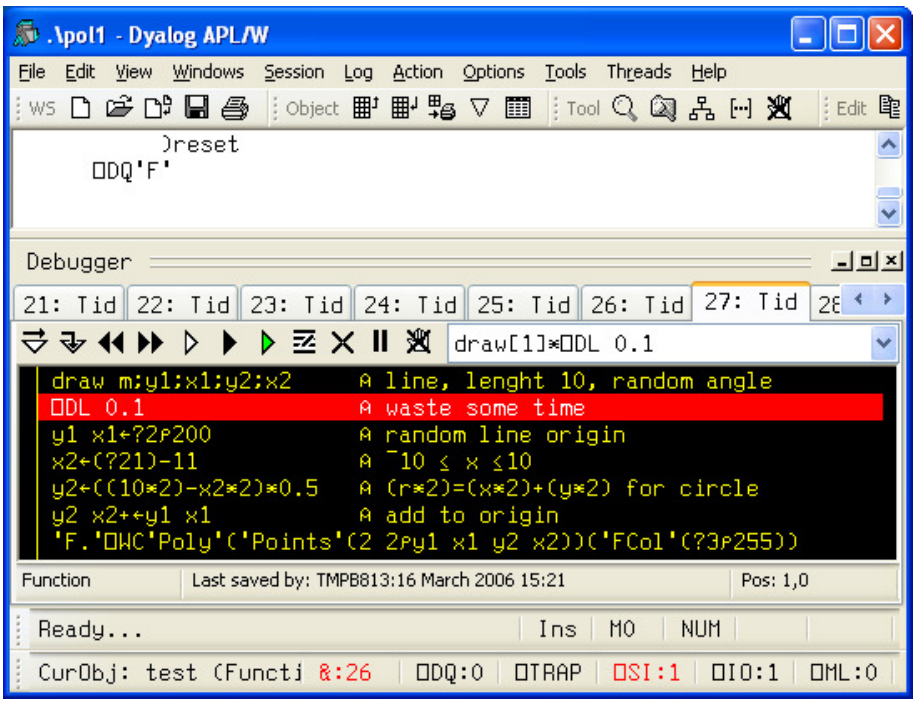
Apply the same procedure again, perhaps with the Threads tool open. Notice the difference in both the drawing rate and the responsiveness of the Session.

Callback functions may be processed by the default Session dequeue mechanism as invoked above, or explicitly using ⎕DQ. ⎕DQ'.' will process events from any active visible object owned by the current thread or created by callbacks from these objects.

13.3.3.2 Give ∇draw∇ a rarg. This can be useful for identifying the owner of an event when tracing. Trace

```
⎕DQ'F'
```

via **Ctrl+Enter**. Move the sprite cautiously into the *Form*. Trace through some of the resulting threads in the Debugger.



A thread may use ⎕NQ to post an event to an object owned by another thread. Any valid Larg except 1 (process immediately) may be used with ⎕NQ&.

13.3.3.3 Pass the Session (or any other window) over the *Form F* to clear the contents (because we used unnamed *Poly* objects which are not refreshed). Enqueue the following events (from a separate thread) and note the effects.

```
      ⎕NQ&'F' 'MouseMove' (?200)(?200) 0 0
      0⎕NQ&'F' 'MouseMove' (?200)(?200) 0 0
      2⎕NQ&'F' 'MouseMove' (?200)(?200) 0 0
      1⎕NQ&'F' 'MouseMove' (?200)(?200) 0 0
DOMAIN ERROR
      F.MouseMove&(?2⍴200),0 0
```

Discuss the fundamental difference between the last two expressions.

For further information on multi-threading in Dyalog APL, see the *Language Reference*, Relnotes.hlp for versions 8.2 and 10.1 and www.dyalog.com [Products][Version 10.1].

13.3.3.4 Please ask for the next module on **TCP/IP Sockets** ☺.