WoWThreads Public Interface

Constants

These constants are found in WowThreads.lua

local L	WoWThreads.L
local sprintf	_G.string.format
local EMPTY_STR	thread.EMPTY_STR
local E	threadErrors
DEBUG	threadErrors.DEBUGGING_ENABLED
SUCCESS	threadErrors.SUCCESS
FAILURE	threadErrors.FAILURE

Signals

These constants may be found in WowThreads.lua

SIG_WAKEUP	The signaled thread continues to execute its while loop.
SIG_RETURN	The signaled thread should exit its while loop, but not necessarily the function.
SIG_TERMINATE	The signaled thread should exit while loop, cleanup state, and return from its function.
SIG_JOIN_DATA_READY	Sent to all threads waiting to retrieve data from the producer thread.
SIG_NONE_PENDING	No signals pending

Functions

These functions are defined in WoWThreads.lua

thread:create()

Creates a new thread of execution.

SIGNATURE: ***local thread_h, result = thread:create(ticks, func [, parameter list])***

PARAMETERS: **ticks**: the time, in clock intervals (a.k.a. ticks), a thread will wait after calling thread:yield(). A tick is equal to the reciprocal of the computer's framerate. At 60 FPS, this equals about 16.7 milliseconds. 60 ticks are about a second.

f: the function the thread is to execute.

parameter list: an optional variable argument list for the thread's function, func

RETURNS: **thread_h**: a handle to the newly created thread

result: a result table containing error information, if any.

EXAMPLE 1:
```
yieldInterval = 60      -- about 1 second
local thread_h, result = thread:create( yieldInterval myFunc )
```

EXAMPLE 2:
```
local thread_h, result = thread:create( 60, printString, "Hello world!")
```

thread:yield()

Instructs the dispatcher to suspend the calling thread for the number of ticks specified when the thread was created (the yieldInterval, see above).

SIGNATURE: ***thread:yield()***

PARAMETERS: None

RETURNS: None

EXAMPLE 1:
```
thread:create()
```

thread:wait()

Instructs the dispatcher to suspend the calling thread for a specified number of clock intervals. This service differs from thread:yield() (above) in that the programmer can delay a thread an arbitrary number of clock intervals.

SIGNATURE: ***thread:wait(waitTime)***

PARAMETERS: **waitTime**: the time expressed in clock intervals the calling thread is to suspend itself.

RETURNS: None

EXAMPLE 1:
```
local waitTime = 30     -- about ½ second
thread:wait( 30 )
```

thread:join()

Suspends the calling thread until the specified thread's data is ready.

SIGNATURE: ***local result = thread:join(thread_h)***

PARAMETERS: **thread_h**: Thread handle of thread for which to wait (a.k.a. the producer thread)

RETURNS: **result**: a result table containing error information, if any.

EXAMPLE:
```
local result = thread:join( producer_h )
```

thread:exit()

This function is called in lieu of return and is used to pass data to threads that have joined the caller. In the example below, the producer thread passes its data to thread:exit(). Internally, thread:exit() sends a SIG_JOIN_DATA_READY signal to all waiting (joiner) threads.

SIGNATURE: ***thread:exit(joinData)***

PARAMETERS: **joinData**: data to be returned to waiting threads

RETURNS: **None**

EXAMPLE:
```
local function producer()
    < do stuff >
   local joinData = getData()
   thread:exit( joinData )
end
local thread_h, result = thread:create( yieldInterval, producer() )
```

thread:self()

Returns the handle and thread Id of the calling thread.

SIGNATURE: ***local thread_h, threadId = thread:self()***

PARAMETERS: joinData: data to be returned to waiting threads

RETURNS: **thread_h**: handle of the calling thread.
threadId: the numerical (unique) Id of the calling thread.

EXAMPLE: *local self_h, selfId = thread:self()*

thread:getId()

Returns the numerical Id of the specified thread. If no thread is specified, then the thread Id of the calling thread is returned.

SIGNATURE: ***local thread_h, threadId = thread:getId([thread_h])***

PARAMETERS: thread_h: handle of the thread whose Id is to be obtained

RETURNS: **threadId**: the numerical (unique) Id of the calling thread.
result: a result table containing error information, if any.

EXAMPLE: *local selfId, result = thread:getId()*
local threadId, result = thread:getId(thread_h)

thread:areEqual()

Returns true if the two threads are identical

SIGNATURE: ***local isEqual = thread:areEqual(thread1_h, thread2_h)***

PARAMETERS: thread1_h: thread handle to be evaluated
thread2_h: thread handle to be evaluated

RETURNS: isEqual: (boolean) true if both threads are the same, false otherwise.

EXAMPLE: *local areEqual = thread:areEqual(thread1_h, thread2_h)*

thread:getParent()

Obtains the handle of the thread that created the specified thread (a.k.a. the parent thread).

SIGNATURE: ***local parent_h, result = thread:getParent([thread_h])***

PARAMETERS: **thread_h**: the thread handle whose parent is to be returned. If not specified, the parent of the calling thread is returned.

NOTE: threads created by Blizzard's WoW client do not have parent threads.

RETURNS: **parent_h**: the parent of the specified thread. If no parent exists (i.e., the specified thread was created by the WoW client, nil is returned.

result: a result table containing error information, if any

EXAMPLE: *local parent_h, result = thread:getParent()*
local parent_h, result = thread:getParent(thread_h)

thread:getChildren()

Obtains the handle of the thread that created the specified thread (a.k.a. the parent thread).

SIGNATURE: ***local childTable, result = thread:getChildren([thread_h])***

PARAMETERS: **thread_h**: handle of the thread whose children (if any) are to be obtained. If not specified, the children of the calling thread are returned.

RETURNS: **childTable**: if the specified thread has one or more child threads, a handle for each child thread is returned in a table of thread handles. If no child thread(s) exist, nil is returned.

result: a result table containing error information, if any

EXAMPLE: *local childTable, result = thread:getChildren()*
local childTable, result = thread:getChildren(thread_h)

thread:getState()

Obtains the execution state of the thread

SIGNATURE: ***local state, result = thread:getState(thread_h)***

PARAMETERS: **thread_h**: handle of the thread whose state is to be obtained. NOTE: by construction, the calling thread is ALWAYS in the "running" state.

RETURNS: **state:** an enumerated set of 3 values: "suspended," "queued," or "completed."

result: a result table containing error information, if any

EXAMPLE: *local state, result = thread:getState(thread_h)*

Error Handling

These two functions are found in threadErrors.lua and MessageFrames.lua, respectively.

threadErrors:setResult(result)

Initializes and returns an error result table.

SIGNATURE: ***local result = threadErrors:setResult(errorMsg)***

PARAMETERS: errorMsg: a string error that message describes the error.

RETURNS: **result**: a result table containing error information, if any. The setResult() function ALWAYS sets the status element to FAILURE and ALWAYS generates a stack trace (via debugstack() at the location where setResult() was called.

```
result = {
        status,
        errorMsg,       -- supplied by the caller
        stackTrace      -- debugstack()
{
```

EXAMPLE:

```
local E = threadErrors
local function someFunction()
    local result = {SUCCESS, EMPTY_STR, EMPTY_STR }
    <do stuff>
    if string1 ~= string2 then
        E:setResult( "string1 ~= string2")
        return status
    end
end
```

mf:postResult()

Displays the contents of a result table in a scrolling text frame.

SIGNATURE: ***mf:postResult(result)***

PARAMETERS: **result:** a result table containing error information.

RETURNS: **None**

EXAMPLE:

```
local E = threadErrors
local result = {SUCCESS, EMPTY_STR, EMPTY_STR }
result = someFunction()
if not result[1] then mf:postResult( result ) return end
```

Debugging Support

These services are found in threadErrors.lua

threadErrors:dbgPrint()

Displays the location, i.e., filename and line number, from where called

SIGNATURE: ***threadErrors:dbgPrint([msg])***

PARAMETERS: **msg:** an optional string.

RETURNS: **none**

EXAMPLE:

local E = threadErrors

E:dbgPrint()

if called in MyFile.lua and line number 85, the following will be printed to the DEFAULT_CHAT_FRAME.

[MyFile.lua:85]

threadErrors:prefix()

Equivalent to dbgPrint() except that prefix() returns the location string. Its intended use is to embed location information in other strings.

SIGNATURE: ***threadErrors:prefix()***

PARAMETERS: **none**

RETURNS: **locationString:** "[Filename:LineNo]"

EXAMPLE:

E = threadErrors

local str = sprintf("%s Hello, world!", E:prefix()

print (str)

[MyFile.lua:85] Hello, world!

Display Service

This function is found in MessageFrames.lua

mf:postMsg()

Display a user/programmer message.

SIGNATURE:	***mf:postMsg(msg)***
PARAMETERS:	**msg:** a user-defined, usually informative, message.
RETURNS:	**None**
EXAMPLE:	*mf:postMsg("Hello world!)*

Management Services

These functions are found in Manager.lua

Mgr:getThreadMetrics()

Calculates a set of metrics for each completed thread from which thread congestion can be calculated. Congestion is calculated as follows:

SIGNATURE:	***congestionTable = mgr:getThreadMetrics()***
PARAMETERS:	**None**
RETURNS:	**congestionTable**: each entry in the table (structure shown below) represents the congestion metric for each completed thread. table = { threadId, -- numerical (unique) Id of the thread. ticksPerYIeld, -- specified at thread creation yieldCount, -- number times thread returned from thread:yield() timeSuspended, -- total time spent in a suspended state threadLifeTime -- elapsed time from creation to completion }
EXAMPLE:	local congestionTable

Thread Congestion – discussion

Congestion is defined as the overhead imposed on a specific thread due to the presence of other threads competing for the system processor. In a perfect world, congestion is the relative ratio of the given by the following formula:

Congestion = [1 – (lifetime with no other threads) / (lifetime with multiple threads)]

The two main variables that determine a thread's congestion are

1. The duration of the yield interval. Short yield intervals lead to higher congestion
2. The number of active threads in the addon. The more threads, the higher the congestion.

A fairly accurate assessment of a thread's congestion can be obtained from the information in the thread's entry in the congestion table. That entry looks like this:

```
local threadId                = entry[1]
local ticksPerYield           = entry[2]
local yieldCount              = entry[3]
local measuredTimeSuspended = entry[4]   -- milliseconds
local measuredLifetime        = entry[5]   -- milliseconds
```

The following procedure calculates the congestion using information from the congestion entry and can be found in the stats:printEntry() function in ThreadStats.lua.

```
function stats:congestion( e )
  local result = {SUCCESS, EMPTY_STR, EMPTY_STR}
      local threadId                = entry[1]
      local ticksPerYield           = entry[2]
      local yieldCount              = entry[3]
      local measuredTimeSuspended = entry[4]    -- milliseconds
      local measuredLifetime        = entry[5]    -- milliseconds

      local meanFramerate = measuredTimeSuspended/(ticksPerYield * yieldCount )
      local totalTicks      = measuredTimeSuspended / meanFramerate
      local congestion      = 1 - (measuredTimeSuspended / measuredLifetime)

      local s1 = sprintf("\n\nThread %d\n", threadId )
      local s2 = sprintf("  time suspended: %.2f ms\n", measuredTimeSuspended)
      local s3 = sprintf("  Lifetime: %d ms.\n", measuredLifetime )
      local s4 = sprintf("  Congestion: %.3f%%\n", congestion * 100 )

      mf:postMsg( s1 .. s2 .. s3 .. s4 )
end
```

In my testing, WoWthreads congestion under normal conditions (< 10 threads, 30 – 50 ticks) seldom rises by about 0.3%.