Mutex

A mutex is a global object that allows you to control code execution order and to prevent conflicts in a multi-thread application.

Basically, a mutex surrounds a portion of JavaScript code, which is then "protected". At runtime, only one JavaScript context can run the protected portion at a time. When a JavaScript context executes the code, it tries to "take" the mutex; if it succeeds, it locks the mutex. However, it will not be allowed to execute the protected code if the mutex was already taken by another context. In this case, it can just ignore the protected code portion or wait until the other context running releases the mutex.

A mutex can be used for example to control a variable value iteration. A mutex also provides a way to pause execution in one thread until a condition is met in another (see the Log example in the **Mutex** section of the *Wakanda Server-Side Concepts* manual).

A mutex is created by the **Mutex( )** constructor method from the Application class.

Mutex Constructor

Mutex( )

Storage **Mutex**(String *key*)

Parameter	Type	Description
key	String	Mutex key
Returns	Storage	New mutex object

Description

The **Mutex( )** method creates a new mutex object that will allow you to control multithreaded concurrent accesses to JavaScript code. For example, you may want to protect a variable value from being modified by a thread B while it is being edited in a thread A. Mutex means *MUTual EXclusion* and designates a lock that can be open or closed.

Pass a string in the *key* parameter. When you call **Mutex( )**, this parameter will act as the key for the mutex object. Any other thread that uses the same key will interact with the same mutex.

A mutex object has three methods:

- **lock( )**
- **unlock( )**
- **tryToLock( )**: this function attempts to lock the mutex and returns true if it could be locked, and false if it could not.

Example

Here is a typical structure for a mutex controlled code:

```
var writeMutex = Mutex('writeMutex');
if (writeMutex.tryToLock())
    {
        //code here executes if tryToLock was able to lock the mutex
        writeMutex.unlock();
    }
    //code here executes even if tryToLock was NOT able to lock the mutex
```

Example

A mutex provides a way to pause execution in one thread until a condition is met in another. In the example below, we want to create a re-usable log file object for use in various server-side functions. To do this, we create a new JavaScript file (named myLog.js in our example) with the following code:

```
function Log(file)     //will use this as the constructor
 {
    this.logFile = null;
    this.init(file);
    return this;
 }

 Log.prototype.init = function(file) //add to the prototype chain of Log
{
    if(typeof file == 'string')     //if we pass a string
        file = File(file);     //assume it to be a path
    this.logFile = file;     //otherwise assume it to be a file
    if (!file.exists)     //if it doesn't exist
        file.create();     //create the empty file
}

 Log.prototype.append = function(message)     //add another function to Log
```

```
{
    if(this.logFile != null){
        //if the Log references a valid file, get a Mutex
        //using the files path as its name
        var logMutex = Mutex(this.logFile.path);
        //attempt to lock the Mutex
        //if it is already locked, the next line pauses execution
        //until it becomes unlocked
        logMutex.lock();
        //from here until logMutex.unlock() we know only one thread
        //can be writing to the log file
        var logStream = TextStream(this.logFile,"write");
        var today = new Date();
        var stamp = today.toDateString() + " " + today.toLocaleTimeString();
        logStream.write(stamp + ': ' + message +"\n");
        logStream.close();
        logMutex.unlock();
    }
}
```

The code above creates a new Log object and then adds two functions to its prototype chain. The first function named init takes either a Wakanda file object or a path to a file as a string. The code then checks that the file exists and if it doesn't, creates it. The next function, named append, takes a string as a parameter. The append function double-checks that the Log object has been initialized (i.e., references a valid log file) and then creates an object by calling **Mutex( )**. In our example we use the log file's path as our key with the goal that only one thread can write to a given file at the same time. We then include the above JavaScript file in our project's code by adding an include statement to our project's main JavaScript file (i.e., projectname.waModel.js) like this:

```
include('myLog.js');
```

This project's main JavaScript file is executed for all calls involving the project so the Log object will be available throughout.

To see an example of the Log object in action we will employ the **verify( )** method, which verifies the data and indices of a Wakanda datastore. Consider the following class method:

```
function verifyData()
{
    var projectLog = new Log('./log.txt');
    //our projects main log file
    var noProblems = true;     //flag to let us know if there were any problems
    //the object below will provide the call back function(s)
    var problemHandler = {
        addProblem: function(problem){
            projectLog.append('Verify: ' + JSON.stringify(problem));
            noProblems = false;
        }
    }
     projectLog.append('Start Verify');
    ds.verify(problemHandler);     //will make periodic call backs
    if (noProblems)
        projectLog.append('Verify found no problems');
    projectLog.append('End Verify');
    return noProblems;
}
```

The **ds.verify** method's single parameter is an object that has functions as attributes. As the verification progresses, Wakanda Server calls back to the appropriate function(s) with information. In the example below, we are only interested in the call back for addProblem, which is called as the verification process discovers individual issues with the data or indices. When addProblem is called, it is provided a problem object containing information about the issue. In our example, we convert the object to a string value and write it to the project's log file after the heading "Verify: " so that when examining the log file later we

can clearly see items concerning this routine. Notice that the log file used is simply “log.txt.” In our project, we use this log file for a variety of tasks and therefore require the services of a mutex to make sure updates to the log are done in an atomic way.

There are several other potential call back methods supported by verify and we could have included these in problemHandler. These include openProgress(title, maxElements), setProgressTitle(title), progress(currentElementNumber, maxElements) and closeProgress(). For more information, see the **verify( )** API.

Mutex Instances

lock( )

Boolean **lock**( )

Returns	Boolean	Always true

Description

The **lock( )** method locks the *Mutex* or wait until it has been released.

If the mutex is unlocked, **lock( )** gets it (returns **true**) and locks it until the **unlock( )** is called or until the script execution ends.

If the mutex is already locked, unlike the **tryToLock( )** method, **lock( )** will pause the code execution until the mutex is released. If several JavaScript threads are trying to lock the same mutex, they will all be paused.

tryToLock( )

Boolean **tryToLock**( )

Returns	Boolean	true if the mutex has been taken and locked, false otherwise

Description

The **tryToLock( )** method tries to lock the *Mutex* or returns **false** if it is already locked.

If the mutex is unlocked, **tryToLock( )** gets it (returns **true**) and locks it until the **unlock( )** is called or until the script execution ends.

If the mutex is already locked, unlike the **lock( )** method, **tryToLock( )** returns **false** and does NOT pause the code execution.

Example

Here is a typical structure for mutex-controlled code:

```
var writeMutex = Mutex('writeMutex');
if (writeMutex.tryToLock())
    {
        //code here executes if tryToLock was able to lock the mutex
        writeMutex.unlock();
    }
    //code here executes even if tryToLock was NOT able to lock the mutex
```

unlock( )

Boolean **unlock**( )

Returns	Boolean	Always true

Description

The **unlock( )** method unlocks the *Mutex* in the current thread and returns **true**.

If the mutex was not locked in the thread, the method does nothing and returns **true**.

If the mutex was locked in the thread, **unlock( )** unlocks it and returns **true**. The mutex can then be taken by other threads.

Example

Here is a typical structure for mutex-controlled code:

```
var writeMutex = Mutex('writeMutex');
if (writeMutex.tryToLock())
    {
        //code here executes if tryToLock was able to lock the mutex
        writeMutex.unlock();
    }
    //code here executes even if tryToLock was NOT able to lock the mutex
```