Multi-Processing with Lua on the Propeller 2 (with Catalina)

Table of Contents

Introduction

Lua is a C-based embedded[1] scripting language that has found popularity in games and other applications where providing the ability for end-users to customize applications easily and rapidly is highly desirable. Lua is not only written in C, it is intended to be highly *inter-operable* with C – you can call Lua programs from C, and vice-versa. But Lua can also be used stand-alone.

Here is an extract from the Wikipedia article about Lua[2]:

> ***Lua*** *(from Portuguese: lua meaning moon) is a lightweight, high-level, multi-paradigm programming language designed primarily for embedded use in applications. Lua is cross-platform, since the interpreter of compiled bytecode is written in ANSI C, and Lua has a relatively simple C API to embed it into applications.*
>
> *Lua was originally designed in 1993 as a language for extending software applications to meet the increasing demand for customization at the time. It provided the basic facilities of most procedural programming languages, but*

[1] The term "embedded" here means embedded within other programs, not embedded in hardware.

[2] See https://en.wikipedia.org/wiki/Lua_(programming_language)

> *more complicated or domain-specific features were not included; rather, it included mechanisms for extending the language, allowing programmers to implement such features. As Lua was intended to be a general embeddable extension language, the designers of Lua focused on improving its speed, portability, extensibility, and ease-of-use in development.*

The Parallax **Propeller** is a parallel processing microcontroller with 8 CPUs, 512 kb of internal RAM and a novel architecture.

Catalina is an ANSI C cross-compiler for the Parallax Propeller, currently supported on Linux and Windows. Catalina originally incorporated Lua purely as a demonstration program for its Catalyst SD-Card based program loader. Catalina supports Lua on both the Propeller 1 and the Propeller 2. On the Propeller 1, which is limited to 32 kb of internal RAM, Lua requires a platform with external RAM (aka XMM RAM) to execute (such as the C3). But *any* Propeller 2 (which has 512 kb of internal RAM) can execute Lua quite handily.

When Catalina needed an embedded scripting capability added to its payload program loader (to allow automated program verification) Lua was the obvious choice. But Lua is basically a single-threaded program, and Catalina has had multi-processing (i.e. multi-threading and multi-cog) capabilities almost from its inception. Most recently (introduced in Catalina 5.0) comprehensive Posix threads support has been added. While this adds no more functionality than already existed, it has made it easier to port multi-threaded applications written in C to the Propeller.

So the question arises – can one add multi-processing support to Lua, to allow it to take advantage of the parallelism of the Propeller? And the answer – fairly obviously – is ***yes!***

There have been many proposals to add multi-processing to Lua, but one – *luaproc* – was originally devised by the author of the Lua language to demonstrate one possible way of doing so. Significantly, it can be completely implemented as a Lua *module* (think *library*) without requiring any modification to Lua itself. Catalina has adopted and extended the proposal in the *luaproc* module into a much more comprehensive *threads* module. However, this module is neither Propeller nor Catalina-specific – it can be used on any platform that has an ANSI C compiler and a Posix threads library[3].

Another reason Catalina chose to adopt the *luaproc* module over various other proposals is that it was based on a *worker* paradigm. To help make C programs capable of exploiting the parallel processing capabilities of the Propeller, Catalina had already introduced a parallelizer program, which used a factory/worker paradigm to simplify adding parallel features to C programs. It was relatively straightforward to extend the *threads* module to add factories, which allows a similar paradigm to be used for Lua programs.

[3] There are some capabilities in the threads module that may require customization to achieve full functionality on other platforms, but the threads module itself should work "out of the box" on any suitable platform. For example, it can be compiled on Windows using the gcc compiler and pthreads library provided as part of mingw.

Note that although Catalina supports Lua on both the Propeller 1 and 2, Multi-processing Lua is only supported on the Propeller 2. The Propeller 1 has insufficient internal RAM to support it. See the Reference section later in this document for more details.

But why use Lua on the Propeller at all?

Mainly because the Propeller is a very interesting architecture with multi-processing capabilities, and Lua is a very interesting and highly extensible language with a small footprint for such a high-level language, but which lacked multi-processing capabilities. It seemed likely that adding multi-processing to Lua could make it possible to exploit the capabilities of the Propeller from a high level language (i.e. instead of only from relatively low-level languages like C, Forth or Spin). Try the tutorial and then judge for yourself whether Multi-processing Lua achieves this.

What this document *is not*, and what it *is!*

This document is ***not*** an introduction to the Parallax Propeller 2. For that, see https://www.parallax.com/propeller-2/#p2-specs.

This document is ***not*** an introduction to Lua. For that, see https://www.lua.org/

This document is ***not*** an introduction to Catalina. For that, see the Catalina documentation such as **Getting Started with Catalina** and the **Catalina Reference Manual**. Catalina is available at https://sourceforge.net/projects/catalina-c/.

This document is ***not*** an introduction to either multi-threading or Posix threads. For that, see any Posix threads documentation. A good place to start would be https://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/91-pthreads/Lib/pthreads-intro.pdf or https://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/91-pthreads/Lib/pthreads-book.pdf

This document ***is*** an introduction to Catalina's ***threads*** module for Lua. It is in the form of both a tutorial with examples and a reference manual. The tutorial assumes you have access to a Propeller 2 board and have the Catalina C compiler (release 5.0 or later) installed.

Lua threads Tutorial

The basics

This section is designed to give you a very quick introduction to what you need to know to get the most out of the Multi-processing Lua tutorials. If you already know some of it (e.g. if you are already familiar with Catalina, Catalyst, Posix threads or Lua) you can simply skim over those sections.

Catalina basics

This document assumes you already have Catalina installed. If not, install it before proceeding, and familiarise yourself with it by reading the tutorial in the document **Getting Started with Catalina**. We will use Catalina in a Catalina command-line window.

Lua itself is a Catalina demo program usually built as part of Catalyst, using either **make** or the **build_all** scripts provided (this is described in the next section) but it is worth becoming familiar with both the **catalina** and **payload** commands.

We will assume you are using a Parallax Propeller 2 Evaluation board, which Catalina knows as the **P2_EVAL**. So, for example, to compile Catalina's "hello world" program, such as you find in the Catalina *demos* folder you would use a command such as:

```
catalina -p2 -lc -C P2_EVAL hello_world.c
```

The `-p2` tells Catalina to compile for the Propeller 2 (the default is to compile for the Propeller 1), `-lc` tells it to link with the standard C library, and `-C P2_EVAL` tells it the board to compile for. You could instead say `-C P2_CUSTOM` if you do not have a Propeller 2 Evaluation board, and make sure that the file *catalina_platforms.inc* in the Catalina folder *target/p2* has the necessary values for your board. Depending on the program being compiled, you might also include options such as `-C COMPACT` to compile your program to use less memory, `-lmc` to link with the floating point library, and (for multi-threaded programs) `-lthreads` to link with the C threads library (which also contains all the Posix threads functions).

To load that program into the propeller and execute it, you would use a command such as:

```
payload hello_world.bin -i
```

The `-i` tells Catalina to start an interactive terminal after loading the program. Other payload options might include `-pX`, where X is the port your Propeller 2 is connected to (if payload does not detect it automatically) or an option like `-b115200` to specify a different baud rate (the default when loading Propeller 2 programs is 230400 baud). You will need to specify all these parameters to use payload as just a terminal emulator without loading a program, such as:

```
payload -i -b230400 -p6
```

Catalyst basics

Lua is typically built as part of **Catalyst**, which is Catalina's SD Card Program Loader, and loaded onto an SD Card for use in the Propeller[4].

[4] If your Propeller 2 does not have an SD Card, you can still execute the interactive version of Multi-processing Lua (i.e. **mlua**) by downloading and starting it in an interactive terminal, using a command like:

```
payload mlua.bin -i
```

You can then copy and paste Lua program code directly into the terminal (which saves typing it all in – but note you should copy source code from the example Lua source files, not from this document!). However, some of the programs will not execute under the interactive **mlua** and instead require **mluax** (which uses less memory because it does not include the language parser/compiler). There is not much

To compile Catalyst and all its associated programs for the Propeller 2, go to the Catalina *demos\catalyst* folder and enter a command like:

```
build_all P2_EVAL COMPACT SIMPLE VT100
```

This builds Catalyst plus all its demo programs – including both Lua and Multi-processing Lua – using a simple serial HMI option (**SIMPLE**) and puts the binaries and demo files in the *demos\catalyst\bin* folder, ready to be copied to an SD Card. You could also specify the `OPTIMIZE` parameter, which will build all the Catalyst programs using the Catalina Optimizer – this makes slightly more memory available to Lua but this compilation can take much longer and is not necessary just to run the tutorial programs.

Note that although Catalyst (and Lua) can use other HMI options, such as a local VGA, keyboard and mouse, we will use a serial HMI option in this document because it is the only HMI option guaranteed to be available on all Propeller platforms.

When compiling Lua, **COMPACT** mode is usually required. While Lua can be compiled and executed in **NATIVE** mode, there is less memory available to run Lua programs. Some of the examples used in the tutorial would be too large to execute without modifications.[5]

The **SIMPLE** and **VT100** options are not required by Lua itself, but **SIMPLE** is required to tell Catalyst to build for a serial HMI (even if this is the default HMI option for the platform) and **VT100** is required when compiling the **vi** text editor.

Copy all the files from *demos\catalyst\bin* to an SD Card, and insert the SD Card into your Propeller platform (make sure to set the switches to boot from the SD Card if required) and start payload communicating with your Propeller connected to port X and acting as an interactive terminal using a command like:

```
payload -b230400 -i -pX
```

Once payload has started, reset the Propeller, and you should see a prompt like:

```
Catalyst 5.0
>
```

You can now enter Catalyst commands.

The commands we will use most frequently in the tutorials will be similar to the following:

To start Multi-processing Lua in interactive mode:

```
mlua
```

that can be done here other than modifying the Lua programs to use less memory – often, this can be done by simply reducing the number of threads etc.

[5] Note that Lua itself can be compiled in **LARGE** or **SMALL** mode to use external memory, but programs compiled in these modes cannot currently be multi-threaded, so Multi-threaded Lua cannot yet be compiled in these modes.

To execute the Lua source program in *example.lua* using Multi-processing Lua in interactive mode:

```
mlua example.lua
```

To edit the file *example.lua* using the **vi** text editor:

```
vi example.lua
```

To compile the file *example.lua* and put the output in *ex.lux* (note that the `-o` option must *precede* the name of the file to be compiled):

```
luac -o ex.lux example.lua
```

To execute the compiled Lua program *ex.lux* using the Multi-processing Lua execution engine:

```
mluax ex.lux
```

In some cases, such as when a program produces more than just a few lines of output, it is useful to use a larger payload window. Try a command like:

```
payload -b230400 -i -g80,60 -pX
```

This will make the payload window 80 columns by 60 lines.

See the **Catalyst Reference Manual** for details of other Catalyst commands (**mv**, **cp**, **ls**, **rm**, **mkdir**, **rmdir**, **dir**, **cat** etc).

NOTE 1: To return to the Catalyst prompt at any time – e.g. if a Lua program locks up or simply never exits – simply reset the Propeller.

*NOTE 2: To exit payload, press CTRL-D **twice**.*

Posix threads basics

Catalina offers comprehensive Posix support in its C *threads* library, and this is what Multi-processing Lua uses. Catalina provides no Posix documentation, but you can refer to the Catalina C header file *include\pthread.h* to see exactly which Posix threads functions Catalina supports, and then refer to any standard Posix threads document for full details.

In addition to Posix threads, Lua makes extensive use of Posix **mutexes** and **condition variables**, and at least a basic understanding of these would be helpful in understanding the tutorial examples. Here is a very brief primer:

A **mutex** is generally used as a means of providing mutually exclusive access to a shared resource (hence the name!). When a thread wants to access the resource, it first attempts to *lock* the mutex, and once it does so, all other threads will be suspended when they try to lock it until the first thread *unlocks* it. Then one *and only one* of the other threads contending for the lock will successfully get it.

A **condition variable** is a means of suspending threads until a specific condition occurs. One or more threads can *wait* for the condition, and they will *all* be suspended until another thread *signals* the condition has occurred. Then *one waiting thread* will be released from suspension. Or, the thread can *broadcast* that the condition has occurred, which will release *all the threads waiting* on that condition – but only one will be given the mutex lock, and must release it to allow the other threads to proceed.

Note the following important points:

1. A **condition variable** is generally used in conjunction with the **mutex** of the same name, and when the condition occurs, one thread will be woken and given the mutex lock, and it must **unlock** the mutex once it has completed whatever processing it is supposed to do as a result of the condition.
2. The condition variable has no *knowledge* of the condition, and holds no *memory* of it – if a thread is not waiting on the condition variable at the time the condition is signalled (or broadcast) but then waits for it, it will be suspended until the *next* time the condition is signalled (or broadcast).
3. A thread can be woken by a signal or broadcast only to discover that by the time it resumes execution the condition is no longer true. It should not *assume* the condition is true – it should *test* it is true (while holding the **mutex** lock) before proceeding. This is one reason why the **mutex** is used – so that only one thread at a time can test the condition.
4. The thread that is woken when a *signal* is sent is not necessarily the *first* one that waited – *any one* of the waiting threads may be woken on a signal. However, *all* waiting threads will be woken when a *broadcast* is sent. But because of the **mutex** lock, only one of the waiting threads will resume execution at a time, with the other thread waiting until they get the mutex lock. Examples of this are given in the tutorial.

Lua basics

The best way to get familiar with Lua is to start the interactive Lua interpreter (we will use the Multi-processing version) and enter some Lua code.

Start the interactive version of Multi-processing Lua from the Catalyst prompt by entering:

```
mlua
```

You should see a Lua banner and prompt, similar to:

```
Lua 5.4.8  Copyright (C) 1994-2012 Lua.org, PUC-Rio
>
```

Traditionally, the first program in any language is a “Hello, World” program. In Lua, this is very simple. Just type in the following line and then press **ENTER**:

```
print "Hello, World"
```

Lua will respond with:

```
Hello, World
```

How easy was that? That one line is the complete Lua program. You could create a file with just that one line in it, compile it with the Lua compiler, and then get Lua to execute the compiled program. In fact, we will do exactly that in the next section.

One thing it is easy to do is confuse the Lua **>** prompt with the Catalyst **>** prompt. But in Lua you can easily *change* the prompt. Enter the following in Lua (note the leading underscore) and press **ENTER**:

```
_PROMPT="LUA>"
```

Now, every time Lua is waiting for input, instead of just **>** it will now prompt with[6]:

```
LUA>
```

Now enter the following lines:

```
a=33
b='a string'
c=false
d=nil
```

We can print these to make sure we entered them correctly. We will use Lua's **print** function. Enter the following:

```
print(a,b,c,d)
```

Lua will respond with:

```
33      a string        false   nil
```

It may not look like it, but this is actually the same **print** function we used in our "Hello, Word" program – as a special case, if the only parameter to a function is a string, Lua allows you to omit the parenthesis. So the following are in fact identical:

```
print "Hello, World"
print("Hello, World")
```

There are several other things worthy of note here:

1. These assignments represent four of Lua's fundamental types:

   a is a **number**, which can be an *integer* or a *float*.

   b is a **string**, which can be delimited using **"** or ' (and there are other delimiters such as **[[** and **]]** specially for use in delimiting multi-line strings)

   c is a **boolean**, which can have the value **true** or **false**.

   d is **nil**, which is actually a special type to Lua. Its main purpose is simply to be different to any other type.

2. We do not have to *declare* variables. We can just *use* them. This is because Lua is a *dynamically typed language*. This means that variables do not have types; only values do. There are no type definitions in the language. All values carry their own type.

Let's try something with our variables. Enter the following (use copy and paste!):

```
if a then print ("TRUE") else print ("FALSE") end
if b then print ("TRUE") else print ("FALSE") end
```

[6] We can do this as part of the **mlua** command, as follows:

```
mlua -e "_PROMPT='LUA>'" -i
```

```
if c then print ("TRUE") else print ("FALSE") end
if d then print ("TRUE") else print ("FALSE") end
```

The responses we will see are:

```
TRUE
TRUE
FALSE
FALSE
```

This may be a little unexpected, since only variable **c** actually contains a **boolean** value. The reason is that Lua considers a variable to be **true** if it has any value other than **false** or **nil**. This can become important in understanding the logic in a Lua program.

Let's try something a little more traditional – defining a simple function to calculate the area of a circle. Enter the following:

```
function area(r)
  return math.pi*r*r
end
```

Now we can *use* our function. Enter the following:

```
print(area(2.5))
```

Lua should respond with:

```
19.63495
```

But wait – where did the value of **pi** come from? As the name implies, it came from something called **math**. This is an example of a **module**. A module can include many things, including constants and functions. For instance, enter the following:

```
print (math.sqrt(4))
```

Lua should respond with the result of calling the **math.sqrt** function:

```
2
```

You can think of modules as libraries. The Lua interpreter automatically opens a standard set of modules so we do not normally need to be too concerned about them. Our **threads** module is one. Our **propeller** module is another. You use the module name to refer to the things (e.g. functions and constants) that it contains. But it is worth noting that one thing you can do with modules is define new names for them. For instance, enter the following:

```
m = math
print(m.pi)
```

Lua will respond with:

```
3.141593
```

A significant feature of Lua is that functions can return more than just a single value. For instance, here is a function that returns *three* values – a string, a number, and a boolean:

```
function tuple(value)
   if type(value) == "number" then
      result=value/2
   else
      result=0
   end
   return "value="..tostring(value), result, value==nil
end
```

First, some notes about the **tuple** function:

1. The **type** function returns the type of the variable as a string – e.g. "number"
2. The **..** operator means string concatenation.
3. The **tostring** function returns its argument converted to type string.

The Lua **print** function will print as many values as are provided as arguments, including all the values returned by any function calls in the argument list. Enter the following:

```
print(tuple(33))
```

This will produce:

```
value=33        16.5    false
```

And if you enter the following:

```
print(tuple("hello"))
```

It will produce:

```
value=hello     0       false
```

If we want to *assign* these return values, we can do so by simply specifying multiple variables in an assignment statement. For instance, enter the following:

```
a,b,c = tuple(nil)
print(a)
print(b)
print(c)
```

This will produce:

```
value=nil
0
true
```

You do not need to assign all the return values. For instance, enter the following:

```
a = tuple(99)
print(a)
```

This will produce:

```
value=99
```

Unused return values are simply discarded, and if we ask for more values to be assigned than are returned, Lua simply assigns **nil** if no corresponding value is returned. For instance, enter the following:

```
a,b,c,d = tuple(66)
print(d)
```

This will produce:

```
nil
```

The final thing we need to know about Lua before we proceed is that there are no traditional structures or arrays in the language. Lua has only **tables**, which are another fundamental Lua type (i.e. along with the "simple" types **numbers**, **booleans**, **strings** and **nil**).

In Lua, a **table** is an *associative array* of keys and values. And the keys do not have to be numbers – they can be *any* Lua type (including **tables**) other than **nil**.

Here is how we might declare a table called T. Enter the following:

```
T = {a = "hello", b = "goodbye", c = 99, 11, 22, 33 }
```

This table has *named* entries that will be known by the keys "a", "b", "c", and Lua will assign consecutive *numbers* to any entries that have not been assigned specific names, such as the last three entries.

Also, in Lua the keys can be specified using *dot notation*, so `T["a"]` is the same as `T.a`. After defining the table above, enter the following:

```
print (T["a"], T.a, T["b"], T.b, T["c"], T.c, T[1], T[2], T[3])
```

This will produce:

```
hello   hello   goodbye goodbye 99      99      11      22      33
```

It is important to understand the relationship between expressions such as `T["b"]`, `T.b` and `T[b]`. The first two refer to the entry of Table T with key "b". The last one refers to the entry of Table T whose key is the *current value* of *variable* b, which in general has *no relationship* to the key "b".

For instance, enter the following:

```
b="a"
print(T["b"], T.b, T[b])
```

This will produce:

```
goodbye goodbye hello
```

Tables are one of the features of Lua that differentiates it from most other languages, and are an extremely powerful feature. We have barely scratched the surface of their importance here, but we now have enough Lua basics to proceed to the main part of this tutorial, which is about the Lua **threads** module, not Lua itself.

Interactive Lua vs compiled Lua

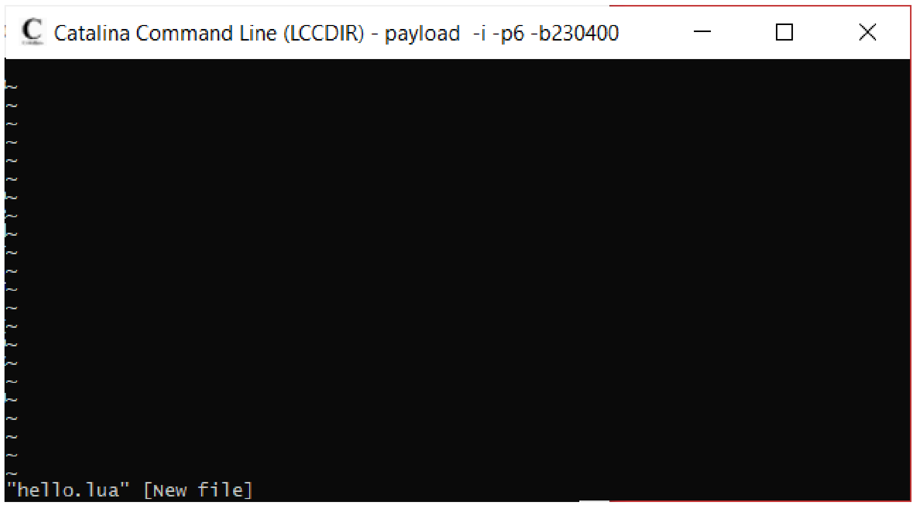
Lua can be very memory-hungry, so when we don't need the interactive capabilities of Lua (e.g. we just want to execute an already compiled Lua program) we can use a Lua execution engine that does not include any of the interactive components (such as the Lua language parser). This frees up more precious memory for the Lua program itself. Some of our example programs will not run using the interactive version of Lua (**mlua**).

The non-interactive version is called **mluax**. Let's use **mluax** to execute our "Hello, World" program.

First, we must create a file containing the Lua program source. We can use Catalyst's **vi** text editor to create or edit it. Enter the following command at the Catalyst prompt:

```
vi hello.lua
```

Assuming the file does not already exist, you will see a screen like the following:

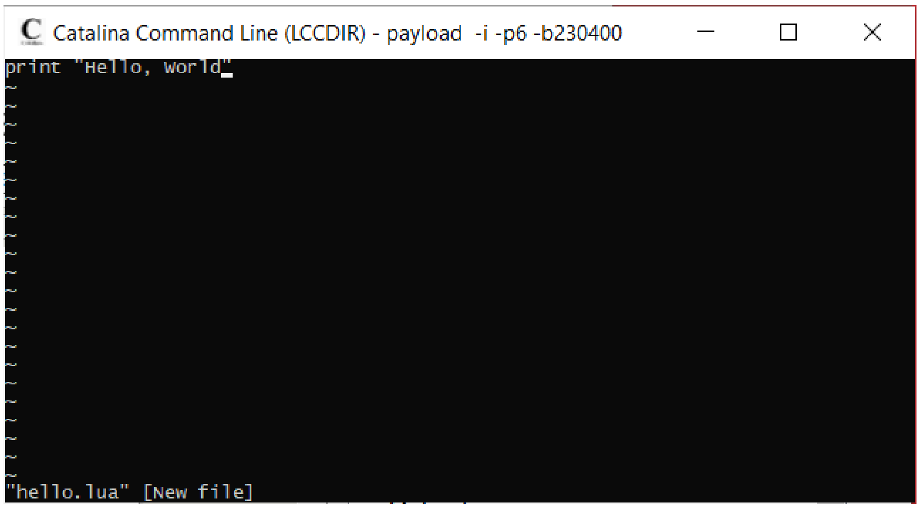

Catalina Command Line (LCCDIR) - payload -i -p6 -b230400

```
~
~
~
"hello.lua" [New file]
```

Type the letter `i` to enter insert mode, and then enter the text of our program:

```
print "Hello, World"
```

Type `ESC` to exit insert mode. Your screen should now look like:


Catalina Command Line (LCCDIR) - payload -i -p6 -b230400

```
print "Hello, World"
~
~
~
"hello.lua" [New file]
```

Type `:wq ENTER` (that's a **colon** followed by **w** followed by **q** followed by **ENTER**) to enter command mode, write the file, and then quit **vi**. You should see the Catalyst prompt return[7].

Next, we can compile the file using the Lua compiler (**luac**). Recall that **luac** can compile both normal and Multi-processing Lua programs. Enter the following at the Catalyst prompt:

```
luac -o hello.lux hello.lua
```

[7] If you get lost in **vi**, try typing **:help** followed by **ENTER**

The compilation should complete without errors. If not, use **vi** again and check you have entered the program correctly.

Next, we will use the non-interactive version of Multi-processing Lua (**mluax**) to execute the compiled program. Enter the following command at the Catalyst prompt:

```
mluax hello.lux
```

You should see:

```
Hello, World
```

Note that you can also use **mlua** to execute either the source version or the compiled version of this particular program. For instance, try:

```
mlua hello.lua
```

```
mlua hello.lux
```

These should both produce the same output. But note that you *cannot* use **mluax** to execute source programs, only compiled ones. For example, If you try:

```
mluax hello.lua
```

You will get an error message:

```
mluax: parser not loaded
```

On the Propeller, the Lua compiler can compile all the example programs used in this tutorial, but the Propeller's limited memory means the compiler may struggle to compile some larger programs. However, the lua compiler can also be compiled for Windows or Linux, and Lua programs can be compiled there for execution on the Propeller.

We now know enough to compile and run all the tutorial programs, so let's get on with it!

Example 1 – A string version of "Hello, World"

You will find a simple *multi-threaded* version of a "Hello, World" program on the SD Card in a file called *ex1.lua*. You can view it using **vi**. Here is what it looks like (with some colour coding added for clarity – in Lua two dashes means the rest of the line is a comment):

```
-- create two workers
threads.workers(2)
-- create a communications channel
threads.channel("ch")
-- create a receiver thread
threads.new('threads.print(threads.get("ch"))')
-- create a sender thread
threads.new('threads.put("ch", "Hello, World from threads")')
-- wait for threads to complete
threads.wait()
```

Now, before we dissect it, let's execute it! To do this we have several options:

1. We could start **mlua** and type the program in manually (not recommended!).

2. We could start **mlua** and then enter the following Lua statement, which (as the name implies) loads a file and then executes the Lua program it contains:

   ```
   dofile "ex1.lua"
   ```

3. We could execute it using **mlua** with a single Catalyst command:

   ```
   mlua ex1.lua
   ```

The last one is the easiest and is recommended for now. But note that some later programs in this tutorial cannot be executed this way – they must be compiled. But we may as well use the easy option in this case.

However we execute it, we should see the output:

```
Hello, World from threads
```

Now let's look at the program itself. Even in this small program there are a few things worthy of note. Here are some of the more significant points:

1. The first statement creates two additional *workers*. We need to be able to execute the sender and receiver threads concurrently for this program to execute correctly – without that, the **put** and **get** function calls cannot complete. The simplest way to ensure this is to create *two* additional workers. Then the program can always run to completion.
2. The second statement creates a message channel (identified as `"ch"`). Note that **send** and **receive** can be used as synonyms for **put** and **get** respectively. A list of such synonyms is given later in this document.
3. The next two statements create the *sender* and *receiver* threads. The *sender* sends a message via the channel `"ch"` to the *receiver*, which just prints out whatever it receives.
4. The receiver uses the function **threads.print** rather than the Lua function **print** because **thread.print** is thread safe. This is not critical in this program (there is only one thread printing) but it will help prevent garbled output in later examples.
5. In this case, since both **put** and **get** functions are synchronous calls which wait until there is a corresponding receiver or sender, we can execute the two **new** function calls in the opposite order and we would see the same output. Edit the program to try it if you want. This demonstrates that the two threads are executing *concurrently* and that the message is not simply being stored somewhere between the execution of the two **new** function calls.
6. Note that the parameter to the **new** function calls are strings containing the Lua code to be executed by the thread. The **new** function accepts either strings or functions. In the next example we will use functions to do the same job, and then we will explain why choosing one method over the other can become important.

7. As an exercise, consider what the program would do if it only created *one* additional worker thread, instead of *two*. Then amend the program and try it. There is a hint to the result in this footnote[8].

Example 2 – A function version of "Hello, World"

Let's look at the next example, which is in a file called *ex2.lua*. It is a slight modification of the previous example:

```
-- create two workers
threads.workers(2)
-- create a communications channel
threads.channel("ch")
-- define a sender function
function sender()
   threads.put("ch", "Hello, World from threads")
end
-- define a receiver function
function receiver()
   threads.print(threads.get("ch"))
end
-- create a receiver thread
threads.new(receiver)
-- create a sender thread
threads.new(sender)
-- wait for threads to complete
threads.wait()
```

The first few lines are identical, but then we define two *functions*, and we use these *functions* in our calls to **new** instead of *strings*. Compare the two programs, and then execute this one with the following Catalyst command:

```
mlua ex2.lua
```

You should see the same output as in the previous example – i.e.:

```
Hello, World from threads
```

Given that it seems to take more setting up, why would you use functions instead of strings? There are two main reasons. The first we can demonstrate immediately. The second one we will identify here, but come back to in the next example:

1. The first reason is that if the Lua code is actually a string, then that string must be parsed at run-time. This means that we have to have the Lua parser available at run-time, or the program will fail. Remember that the **mluax** program does not include the parser, so while both examples can be executed with **mlua**, only this one can be executed with **mluax**. Try compiling both versions using **luac**, and then executing them with **mluax**, as follows.

8 Hint: One worker can execute more than one thread. If the thread that is currently executing suspends itself for some reason and there is another thread waiting to execute, then the worker will begin executing the other thread. But it is better for threads not to rely on this behaviour unless there is no alternative, since it depends on the other executing threads suspending themselves. For further discussion on the situations in which threads are suspended, see example 11.

First, compile them both, using the following Catalyst commands:

```
luac -o ex1.lux ex1.lua
luac -o ex2.lux ex2.lua
```

Now use **mluax** to execute the function version (i.e. example 2). Enter the following Catalyst command:

```
mluax ex2.lux
```

You should see the expected output:

```
Hello, World from threads
```

So far so good! But then use **mluax** on the string version (i.e. example 1). Enter the following Catalyst command:

```
mluax ex1.lux
```

And you will see:

```
mluax: ex1.lua:6: parser not loaded
stack traceback:
[C]: in function 'new'
ex1.lua:6: in main chunk
[C]: ?
```

Ouch! We cannot use **mluax** to execute programs containing **new** calls with string parameters because it has no parser. And this can be important because we may not be able to spare the memory required to load the parser at run-time.

2. The second reason is that if the Lua code is actually a string, then it is very difficult to *parameterize* it – something that we can easily do with functions. For example, suppose we wanted *generic* senders and receivers that could send different messages or use different channels. This would be complex and messy to do with strings, but we can easily generate a function to do this, which is our next example.

Note that the parameter to the **new** function is the *name* of a function – e.g. `sender`, not the result of *invoking* a function – e.g. `sender()`. Also note that this means that a function used to create a thread cannot accept any parameters – but see the next example which illustrates how to get around this apparent limitation.

Example 3 – A generic version of "Hello, World"

Let's look at the next example, which is in a file called *ex3.lua*. It is a slight modification of the previous example:

```
-- create two workers
threads.workers(2)
-- create two communications channels
threads.channel("ch1")
threads.channel("ch2")
-- define a sender function generator function
function sender(ch, msg)
   return function()
      threads.put(ch, msg)
   end
```

```
end
-- define a receiver function generator function
function receiver(ch)
   return function()
      threads.print(threads.get(ch))
   end
end
-- create a receiver thread
threads.new(receiver("ch1"))
-- create a sender thread
threads.new(sender("ch1","Hello, World from threads"))
-- create another receiver thread
threads.new(receiver("ch2"))
-- create another sender thread
threads.new(sender("ch2","Hello Again, World"))
-- wait for threads to complete
threads.wait()
```

If you execute this example with the Catalyst command:

```
mlua ex3.lua
```

You will see:

```
Hello, World from threads
Hello Again, World
```

First, note that we only create *two* workers, but the program creates *four* Lua threads in total (two *senders* and two *receivers*). This is fine because Lua will automatically queue the execution of the second pair of threads to start once the workers become available.

Next, examine the new *sender* and *receiver* functions. These functions are not the ones executed directly by the workers – instead they *return* the functions that are executed by the workers, which are generated using the given parameters. This is possible because Lua has what are called *first-class* functions. This means that functions are another type of value in Lua, and can be returned as the *result* of a function.

In this example, the *sender* function accepts the name of the channel and the message to send, and generates an anonymous function that will send that message on that channel, and the *receiver* function accepts the name of a channel and will generate a new anonymous function that will receive a message on that channel and print it.

First-class functions are another powerful feature of the Lua language. A full explanation of this language feature is beyond the scope of this tutorial, but this example illustrates how the feature can be used as *parameterized* or *generic* functions, which is very handy when creating threads.

Example 4 – Using Message Channels

We have already used message channels in the previous examples, but in this example, we will illustrate how message channels can be used to both communicate between and synchronize threads. You will find the following program in the file *ex4.lua*:

```
t = require "threads"
-- we want two simultaneous threads
t.workers(2)
-- we want two message channels
t.channel("ping")
t.channel("pong")
-- we use one shared global variable
t.update("count",21)

-- one thread "pings"
function ping()
  t = threads
  count = t.shared("count")
  for i = 1, count do
    ball = t.get("ping")
    t.output("ping ")
    t.put("pong", ball)
  end
end

-- the other thread "pongs"
function pong()
  t = threads
  count = t.shared("count")
  for i = 1, count do
    ball = t.get("pong")
    t.output("pong\n")
    if (i < count) then
      t.put("ping", ball)
    end
  end
end

-- create the threads
t.new(ping)
t.new(pong)
-- kick things off
t.put("ping", "ball")
-- wait for threads to complete
t.wait()
```

Can you guess the output it will produce just by examining the code? Execute it with the following Catalyst command to check your guess:

```
mlua ex4.lua
```

Here are some things to note about this example:

1. The first line creates the variable `t` as a synonym for the `threads` module. We could also have just said `t = threads`. This saves us some typing, but note that we also have to use a line like `t = threads` in every new Lua state, such as in each of the functions.
2. We use one shared variable (called `count`) to share information between the main thread, the sender and receiver. We use the shared variable functions **update** to set its value, and **shared** to fetch it. We will learn more about shared global variables in the next example.
3. The function `pong` must avoid doing a final **put**, because it will never be matched by a **get**, which means it would never return and the program would never terminate.

Example 5 – Using Shared Global Variables

We have already used a shared global variable in the previous examples, but in this example, we will use *only* shared variables. You will find the following program in the file *tex5.lua*:

```
t = require "threads"
-- we want two simultaneous threads
t.workers(2)
-- we use two shared global variables
t.update("count", 21)
t.update("turn", nil)

-- one thread "pings"
function ping()
  t = threads
  count = t.shared("count")
  while count > 0 do
    if t.shared("turn") == "ping" then
       t.output("ping ")
       t.update("turn", "pong")
       count = count - 1;
    end
  end
end

-- one thread "pongs"
function pong()
  t = threads
  count = t.shared("count")
  while count > 0 do
    if t.shared("turn") == "pong" then
       t.output("pong\n")
       t.update("turn", "ping")
       count = count - 1;
    end
  end
end
```

```
-- create the two threads
t.new(ping)
t.new(pong)
-- set the shared variable to kick things off
t.update("turn", "ping")
-- wait for threads to complete
t.wait()
```

The output of this program will be the same as the previous example. Execute it with the following Catalyst command to check:

```
mlua ex5.lua
```

This program uses the function **update** to set a shared variable value, and **shared** to fetch it. We could also first set a local variable to the value we need and then use **export** to share the variable, but **update** is easier when dealing with simple values, especially if we don't need to store the value locally. Also note that **export** can also be used to share *tables*, whereas **update** can only be used to set *simple values* (i.e. nil, booleans, numbers or strings).

The reason that shared variables are useful is that normal Lua variables only exist only in a single Lua state – so if the main thread just used the variable **count** and so did the sender and receiver, they would all be using *different* instances of the variable. This is because the main thread, the sender and the receiver are executing in different Lua *states*.

The threads package ensures all access to shared variables is *thread-safe*, but if we are dealing with more complex data than a single simple variable (such as a set of variables or a table) then we may need to use a **mutex** to ensure that only one thread can update the shared data at once. Otherwise, we could end up with a *race condition*[9].

While this example works, note that the threads in this program are "busy waiting" – i.e. the threads are constantly checking the shared variable `"turn"` to see if they should print. While shared variables are a good way to share information between threads, they are not recommended as a synchronization mechanism. Fortunately, we also have *condition variables*, which we will demonstrate in the next example.

Example 6 – Using Condition Variables

Condition variables provide a more sophisticated synchronization mechanism than shared variables. You will find the following program in the file *ex6.lua*:

```
t = require "threads"
-- we want four simultaneous threads
t.workers(4)
-- here is how we will identify them, which
-- is also a string for each one to output
eeny  = "eeny "
```

[9] A *race condition* occurs when the result of the program depends on the *order* in which threads execute or the way in which concurrent threads *interleave* their execution. It makes the outcome of the program difficult to predict, bugs difficult to track down, and is generally undesirable.

```
meeny = "meeny "
miney = "miney "
mo    = "mo\n"
-- we need four condition variables
-- (one for each thread)
t.condition(eeny)
t.condition(meeny)
t.condition(miney)
t.condition(mo)
-- generate a thread function
function Thread(me, next, count, last)
   return function()
     t = threads
     repeat
        t.lock(me)
        t.update(me, true)
        t.wait(me)
        t.update(me, false)
        t.unlock(me)
        t.output(me)
        count = count - 1
        if (count > 0) or not last then
           next_ready = false
           while not next_ready do
              t.lock(next)
              next_ready = t.shared(next)
              t.unlock(next)
           end
           t.signal(next)
        end
     until count == 0
   end
end
-- wait till a thread is "ready"
-- (i.e. waiting on its condition variable)
function wait_till_ready(this)
   local this_ready = false
   while not this_ready do
      t.lock(this)
      this_ready = t.shared(this)
      t.unlock(this)
   end
   return this_ready
end
- number of iterations
count = 20
-- create the threads
t.new(Thread(eeny,  meeny, count, false))
t.new(Thread(meeny, miney, count, false))
t.new(Thread(miney, mo,    count, false))
t.new(Thread(mo,    eeny,  count, true))
-- wait till eeny is "ready"
wait_till_ready(eeny)
-- kick things off
```

```
t.signal(eeny)
-- wait for threads to complete
t.wait()
```

Note that you may need to compile this program with **luac** and execute it with **mluax**. If the following Catalyst command does not work:

```
mlua ex6.lua
```

Then try the following instead:

```
luac -o ex6.lux ex6.lua
mluax ex6.lux
```

Here is what the output should look like:

```
 eeny meeny miney mo
 eeny meeny miney mo
 eeny meeny miney mo
  ... (etc) ...
```

This version solves the main problem with the previous example – threads are no longer spending most of their time "busy waiting". While waiting on a condition variable, a thread (nominally, although this depends on the implementation) consumes no processor time.

Note that before waiting on a **condition variable**, a thread must first lock the corresponding **condition mutex** (which will have the same name). The **wait** *unlocks* the mutex, so that other threads can also lock the mutex or wait on the same condition variable – but when the wait completes, the mutex will be *locked* again before any waiting thread is allowed to continue. The thread should *unlock* it again as soon as practical.

This can be done using code similar to the following (for a **condition** called `cond`):

```
t.lock(cond)             -- see 1, below
t.update(cond, true)     -- see 2, below
t.wait(cond)             -- see 3, below
t.update(cond, false)    -- see 4, below
t.unlock(cond)           -- see 5, below
```

Note that we have introduced a **shared variable** with the same name as the **condition variable** (i.e. `cond`)[10] - this is recommended. Here are some notes about the above code:

1. It *locks* the **condition mutex**;
2. The **shared variable** is set to true, to indicate that this thread is about to wait on the **condition variable**. Changes to the shared variable must only be done while the **condition mutex** is *locked*, so that the variable accurately represents the condition;
3. A wait is performed on the **condition variable**. Note that the **condition mutex** is automatically *unlocked* during the wait (which allows other processes to claim the lock as and when required);

[10] Note this means there are now *three* objects with the same name, each with specific operations - i.e. the **condition variable**, the **condition mutex** and the **shared variable**.

4. The **shared variable** is set to false, to indicate that the thread is no longer waiting on the **condition variable**. Note that the **condition mutex** is automatically *re-locked* before the wait returns;
5. It *unlocks* the **condition mutex**.

To determine if the condition has occurred, code similar the following should be used:

```
ready = false
while not ready do          -- see note 1, below
   t.lock(cond)             -- see note 2, below
   ready = t.shared(cond)   -- see note 3, below
   t.unlock(cond)           -- see note 4, below
end
t.signal(cond)              -- see note 5, below
```

Here are some notes about the above code:

1. A thread needing to determine if a condition has occurred should only proceed if the corresponding **shared variable** is true - in this code, it loops until this is the case;
2. It *locks* the **condition mutex** to ensure that the **shared variable** cannot be changed while it is determining whether or not the condition has occurred;
3. It retrieves (and remembers) the value of the **shared variable**;
4. It *unlocks* the **condition mutex**;
5. If the **shared variable** was true, then it means a thread that uses the *previous* code is now *guaranteed* to be waiting on the **condition variable**, so the thread that uses *this* code can safely signal that the event has occurred. Without such a guarantee, a signal might be sent when no thread is waiting for it, and any such signals are simply lost.[11]

It is the **shared variable** that actually represents the “condition” associated with the **condition variable**. In this case, the “condition” is simply that *a thread is waiting for that signal*, but it could be a more complex condition. However, even for this simple condition a separate boolean variable is required because the condition *cannot be deduced* from the **condition variable** itself.[12] If there were to be more than one thread possibly waiting on the condition, the code would need to be more complicated, and perhaps use a count as the **shared variable** rather than a simple boolean. Also, strictly speaking, when a thread wakes, it should *check* that the condition it was expecting has *actually* occurred (and not just assume that it has) because on most systems it is possible for threads to be woken spuriously, or for other reasons (e.g. because an unrelated operating system event has occurred).[13]

Failure to follow procedures like those above when using condition variables can lead to unexpected or undefined behaviour - the code may work on some platforms but not others.

[11] This is not a limitation of Catalina’s implementation - it is inherent in the design of condition variables.
[12] Ditto.
[13] Ditto.

Finally, note that a thread must not do a coroutine **yield** or a thread **put** or a **get** while the condition mutex is locked – this limitation is discussed further in example 11.

Example 7 – Asynchronous Send and Receive

We have already used the **send** and **receive** functions in previous examples (recall that they are synonyms for **put** and **get**), but only the synchronous versions. This example illustrates the use of both *synchronous* and *asynchronous* versions. You will find the following program in the file *ex7.lua*:

```
t = require "threads"
-- we want two workers
t.workers(2)
-- we want one message channel
msg = "msg"
t.channel(msg)

-- sender sends synchronously until it succeeds
-- (or retries 10 times):
function sender()
  t = threads
  local res
  local retries = 0
  repeat
    t.sleep(3)
    t.print("sending synchronously")
    res = t.send("msg", "'string'", 2, nil, true)
    if not res then
       t.print("error sending")
    end
    t.sleep(1)
    retries = retries + 1
  until res or retries == 10
end

-- asender sends asynchronously until it succeeds
-- (or retries 10 times):
function asender()
  t = threads
  local res
  local retries = 0
  repeat
    t.sleep(1)
    t.print("sending asynchronously")
    res = t.send_async("msg", "'string'", 2, nil, false)
    if not res then
       t.print("error sending")
    end
    t.sleep(1)
    retries = retries + 1
  until res or retries == 10
end
```

```
-- receiver receives synchronously until it succeeds
-- (or retries 10 times):
function receiver()
  t = threads
  local res1, res2, res3, res4
  local retries = 0
  repeat
    t.sleep(3)
    t.print("receiving synchronously")
    res1, res2, res3, res4 = t.receive("msg")
    if not res1 then
       t.print("error receiving")
    else
       t.print("result is ", res1, res2, res3, res4)
    end
    retries = retries + 1
    t.sleep(1)
  until res1 or retries == 10
end

-- areceiver receives asynchronously until it succeeds
-- (or retries 10 times):
function areceiver()
  t = threads
  local res1, res2, res3, res4
  local retries = 0
  repeat
    t.sleep(1)
    t.print("receiving asynchronously")
    res1, res2, res3, res4 = t.receive_async("msg")
    if not res1 then
       t.print("error receiving")
    else
       t.print("result is ", res1, res2, res3, res4)
    end
    retries = retries + 1
    t.sleep(1)
  until res1 or retries == 10
end
```

Then there are various attempts to send and receive, using all the possible combinations of synchronous and asynchronous senders and receivers.

The first three combinations will succeed (sometimes after retying) but the last one will not:

```
-- this will succeed without retries…
t.print("\nSynchronous send and Synchronous receive:")
t.print("(should succeed on first attempt)\n")
t.new(receiver)
t.new(sender)
t.wait()

-- this will succeed after retrying …
```

```
t.print("\nSynchronous send and Asynchronous receive:")
t.print("(should succeed after some retries)\n")
t.new(areceiver)
t.new(sender)
t.wait()

-- this will succeed after retrying …
t.print("\nAsynchronous send and Synchronous receive:")
t.print("(should succeed after some retries)\n")
t.new(receiver)
t.new(asender)
t.wait()

-- this will probably never succeed …
t.print("\nAsynchronous send and Asynchronous receive:")
t.print("(will probably never succeed)\n")t.new(areceiver)
t.new(asender)
t.wait()
```

You can execute this program with the following Catalyst command:

```
mlua ex7.lua
```

Here is what the output will look like:

```
Synchronous send and Synchronous receive:
(should succeed on first attempt)

receiving synchronously
sending synchronously
result is       'string'        2       nil     true

Synchronous send and Asynchronous receive:
(should succeed after some retries)

receiving asynchronously
error receiving
receiving asynchronously
error receiving
sending synchronously
receiving asynchronously
result is       'string'        2       nil     true


Asynchronous send and Synchronous receive:
(should succeed after some retries)


sending asynchronously
error sending
receiving synchronously
sending asynchronously
result is       'string'        2       nil     false

Asynchronous send and Asynchronous receive:
(will probably never succeed)
```

```
receiving asynchronously
error receiving
sending asynchronously
error sending
receiving asynchronously
error receiving
```

... etc …

Example 8 – Many Concurrent Threads

Our next example in this tutorial simply demonstrates how many concurrent Lua threads the Propeller 2 can execute.

You will find the following program in the file *ex8.lua*:

```
t = threads
-- 10 threads assumes Lua is compiled in COMPACT mode.
-- If it is compiled in NATIVE mode change count to 5:
count = 10
-- we only need a small stack
t.stacksize(2500)
-- create the workers
t.workers(count)

-- this function generates a thread function
function Thread(me, iter)
   return function()
     t = threads
     m = require 'math'
     -- wait till we are told to go
     repeat until t.shared("go")
     repeat
        t.output(me .. " ")
        iter = iter-1
        -- a small delay here makes it more evident
        -- we are using pre-emptive multitasking
        t.msleep(m.random(10))
     until iter == 0
   end
end
-- create the threads
for id = 1,count do
   -- collect garbage to maximize memory
   collectgarbage()
   t.new(Thread(id, 100))
   t.print(t.sbrk(true))
end

-- kick things off
go = 1
t.export("go")
t.wait()
```

Note that you need to compile this program with **luac** and execute it with **mluax** or it will not have enough memory to start all the threads. Enter the following Catalyst commands:

```
luac -o ex8.lux ex8.lua
mluax ex8.lux
```

Here is what the output should look like (note that the precise output may vary on your Propeller 2):

```
331744
346208
361184
376096
391008
405920
420832
435744
450656
465568
9 8 7 6 5 4 3 9 2 1 8 6 7 5 3 2 9 4 1 5 8 2 6 3 7 4 9 1 8 2 5 3 6 4 2 7 1 9 8 5
3 4 6 1 9 2 8 7 6 5 3 8 4 2 1 9 7 6 5 3 8 6 4 2 7 5 1 9 3 8 2 4 5 7 1 6 9 10 8 4
 3 2 1 6 7 5 9 10 8 2 3 4 7 6 1 5 9 10 8 3 2 4 9 7 6 1 5 10 4 8 6 3 2 7 9 5 1 10
 4 6 3 2 8 5 7 9 6 4 1 3 10 2 8 5 4 3 1 9 7 6 10 2 5 8 4 9 10 2 7 5 6 3 1 4 9 8
10 2 7 5 3 1 6 8 4 7 9 2 3 5 10 1 6 8 7 9 4 5 3 2 1 10 6 9 5 4 8 7 2 3 1 10 6 9
4 5 2 1 8 7 3 10 4 6 5 9 2 7 3 10 8 6 1 5 4 2 7 9 10 3 4 1 8 5 6 9 2 7 10 3 8 5
1 4 6 9 7 10 2 3 8 5 10 2 1 4 8 6 9 7 3 5 2 10 8 6 4 3 1 9 5 7 2 10 8 3 9 5 1 6
7 4 3 10 2 8 9 1 5 10 6 7 4 3 2 8 5 9 1 4 6 7 3 10 2 8 4 5 9 3 6 1 7 10 2 4 5 8
9 1 6 8 3 7 10 4 2 5 9 6 7 1 8 10 3 7 4 2 5 9 1 6 8 3 10 7 5 2 1 9 4 6 7 10 5 8
1 3 2 9 4 6 7 5 1 10 8 4 2 3 9 7 6 5 4 1 2 8 6 3 10 7 9 5 2 4 1 7 6 5 8 3 9 10 2
 1 4 5 7 6 9 10 8 3 2 1 5 7 4 10 6 2 1 9 8 3 5 7 4 1 9 6 10 2 3 8 5 7 1 10 6 4 9
 2 3 7 8 5 4 10 6 9 1 3 5 2 10 8 7 6 9 4 1 8 10 3 5 7 2 6 1 3 4 9 8 10 7 5 2 6 3
 1 4 8 5 9 7 2 10 1 6 4 8 3 7 5 9 2 10 4 8 6 3 1 7 2 9 5 4 10 3 8 7 6 1 2 4 9 10
 5 1 6 3 8 10 7 4 5 2 9 1 6 3 7 4 8 2 10 5 1 9 3 6 4 8 2 10 7 5 3 6 4 2 9 1 10 8
 7 3 4 5 6 2 9 8 7 3 10 1 5 6 4 2 3 1 9 8 10 7 6 4 5 2 3 9 10 1 8 7 6 2 4 5 9 3
10 7 1 2 4 8 6 5 9 1 2 7 3 10 8 4 9 6 5 1 10 8 2 7 3 9 4 5 6 1 2 10 7 9 8 3 2 1
5 4 6 7 9 3 2 10 1 6 8 5 4 7 9 3 10 1 6 5 7 8 2 9 3 4 10 8 5 6 7 1 2 4 3 9 6 10
7 8 5 1 9 2 4 6 3 10 7 8 5 1 9 10 7 2 3 5 4 8 6 1 9 2 10 3 7 4 8 6 5 1 10 2 9 3
8 4 7 5 6 1 10 9 2 7 4 3 5 8 6 1 10 9 7 3 5 4 2 8 1 6 10 3 7 5 9 4 1 8 2 6 5 10
7 3 4 9 2 1 8 6 7 4 5 10 2 9 8 3 7 6 1 4 5 10 2 8 7 9 5 1 4 3 2 6 10 8 1 5 7 9 3
 2 4 6 1 8 10 7 5 3 2 9 4 1 8 10 6 3 2 9 7 5 4 1 10 8 6 5 7 3 2 9 10 4 1 5 6 3 8
 2 4 9 10 7 1 5 6 3 9 8 2 10 5 4 7 1 8 6 3 2 9 5 7 1 10 4 9 3 8 6 2 10 7 4 5 1 9
 8 3 6 2 1 7 4 5 10 9 6 8 1 3 5 4 10 9 7 2 6 8 3 1 4 5 2 7 8 9 1 6 4 10 3 7 5 2
1 8 4 10 3 9 6 7 2 4 8 1 6 5 10 3 9 7 2 1 10 6 5 8 4 3 9 7 2 6 4 1 10 8 2 9 7 5
3 4 6 10 8 2 1 9 7 4 5 3 10 2 6 9 8 4 1 7 10 5 3 2 6 9 8 1 4 7 5 3 10 1 9 8 2 6
7 5 4 10 1 8 2 3 9 4 6 7 1 10 6 8 3 7 4 9 10 1 6 3 8 1 10 9 6 3 8 6 10 3 9 8 10
9 10 10 10 10 10 10
```

The first 10 numbers[14] are the output of **sbrk** function, which shows the current C heap just after each of the 10 threads is started. Given that the Propeller 2 has 512k bytes of Hub RAM, and that some of the upper Hub RAM is used for other things, when this

[14] If Lua was compiled in **COMPACT** mode, 10 threads should be possible. If it is compiled in **NATIVE** mode, the number has to be reduced – 5 should be possible.

number approaches 500000 it means the program is using nearly all the Propeller Hub RAM.

Note the use of the Lua garbage collector – this helps to maximise the available memory before creating the threads. There is more about Lua memory management in example 9, and also in the technical notes at the end of this document.

Example 9 – Rendezvous, Recycling, Factories and More!

Our next example demonstrates a few of the features we have not used yet – i.e. rendezvous, recycling worker states, and multiple factories. It also demonstrates various techniques that can be used to reduce Lua's memory usage.

You will find the following program in the file *ex9.lua.* It is a variation of example 6. The notes below the program tell you what you need to know about it, and what you need to observe when it is executed:

```
-- set up garbage collection ...
lua_version = tonumber(string.sub(_VERSION,5,7));
if lua_version and lua_version < 5.4 then
  -- default mode is incremental, so just set parameters ...
  collectgarbage("setpause", 150);
  collectgarbage("setstepmul", 500);
elseif lua_version and lua_version < 5.5 then
  -- incremental mode is optional, so set it with parameters ...
  collectgarbage("incremental", 150, 500);
else
  -- mode and parameter setting now separate in Lua 5.5 ...
  collectgarbage("incremental");
  collectgarbage("param", "pause", 150);
  collectgarbage("param", "stepmul", 500);
end

t = require "threads"

-- we want four factories
t.factories(4)
-- we want four simultaneous threads
t.workers(4)
-- we want to recycle all our workers
t.recycle(4)

-- here is how we will identify the threads, which
-- is also a string for each one to output
eeny  = "eeny "
meeny = "meeny "
miney = "miney "
mo    = "mo\n"

-- we need four condition variables
-- (one for each thread)
t.condition(eeny)
t.condition(meeny)
t.condition(miney)
```

```
t.condition(mo)

-- generate a thread function
function Thread(me, next, count, last)
   return function()
     t = threads
     f = t.factories()
     repeat
        -- first, act as consumer
        t.rendezvous(me)
        t.output(t.factory(), ":", me)
        t.factory((t.factory() % f) + 1)
        -- then act as producer, unless we are
        -- the last thread in the last round
        if not last or (count > 1) then
           t.rendezvous(next)
        end
        count = count-1
     until count == 0
     -- collect garbage
     t.gc(0);
     t = nil
   end
end

count = 5
iteration=0
-- print current top of heap
t.print("Heap = ", t.gc(1))

-- iterate to show the effect on memory usage
while iteration < 100 do
   iteration = iteration + 1
   t.print("\nIteration ", tostring(iteration))
   -- create the threads
   NewThread(eeny,  meeny, count, false)
   NewThread(meeny, miney, count, false)
   NewThread(miney, mo,    count, false)
   NewThread(mo,    eeny,  count, true)
   -- kick things off
   t.rendezvous(eeny)
   -- wait for threads to complete
   t.wait()
   -- wait a short time between iterations
   t.msleep(100)
   -- collect garbage, defragment the heap
   -- and print current top
   t.print("Heap = ", t.gc(1))
end
```

You may need to compile this program to execute it[15]. You can do so by entering the following Catalyst commands:

```
luac -o ex9.lux ex9.lua
mluax ex9.lux
```

This program creates 4 workers in 4 factories, uses rendezvous to synchronize the threads, and it also recycles the workers between iterations.

A **rendezvous** is nothing special. It is really just a convenience function that implements the following logic (which is often used in multi-threaded programs such as the classic producer/consumer problem) in a single call:

```
lock(condition)
broadcast(condition)
wait_for(condition)
unlock(condition)
broadcast(condition)
```

It is worth considering what happens when *two* threads execute the above logic on the same condition – it turns out that it does not matter in which order the threads execute, or how their execution is interleaved, both threads will be held until both have completed executing this logic. Then they are both allowed to proceed[16].

Rendezvous are convenient because they can eliminate the possibility of race conditions among competing threads.

When you execute this program, each thread will not only print its message, it will include the factory number in which the Lua thread is currently executing. For instance, you will see lines like:

```
1:eeny 3:meeny 4:miney 2:mo
```

This means thread **eeny** is executing on factory 1, **meeny** is executing on 3, **miney** is executing on 4, and **mo** is executing on 2.

Remember that a factory is essentially a cog, so we are now executing our Lua program on 4 cogs concurrently, and all we had to do to enable this was add one function call at the beginning of the program – i.e:

```
t.factories(4)
```

You can edit this line and alter the number of factories to 1, 2 or 3 or 5 (assuming you have enough free cogs[17]) to see the effect it has.

[15] Even when compiled, this program may not have enough memory to run if Lua is compiled in NATIVE mode. In that case, try reducing the number of workers that are recycled – e.g. try `threads.recycle(2)`.

[16] For those familiar with Posix threads, a **rendezvous** is similar to a two-thread barrier.

[17] If Lua is compiled with the **SIMPLE** HMI option then five cogs should be available. Other HMI options may leave less cogs available, but if there are insufficient cogs then the program will still execute, but increasing the number of factories beyond the number of available cogs will have no effect.

Note also that the threads themselves use the **factory** function to *move* from factory to factory (i.e. from cog to cog!). This is done by the following line in the middle of the thread function:

```
t.factory((t.factory() % f) + 1)
```

The main program loop is executed 100 times to show how the memory usage behaves (e.g. that it grows fast initially and then stabilizes). This program thus creates threads 400 times. Thread creation is an expensive process in Lua, but we can use Lua's garbage collector to help reduce the use of memory. The various memory management techniques used in this program are discussed further below, and also in the Technical Notes at the end of this document.

When the program executes, observe the following:

The Lua threads are allocated to different factories automatically on startup. Each worker is automatically allocated to a different factory (if available) by the **new** function, so if there are 4 or more factories then each worker ends up in a factory of its own.

The current top of the C heap is printed before and after each iteration (using the **gc** function. It can take a few iterations for the heap usage to stabilize, but it will.

Next, edit the file and comment out (or delete) the line `t.recycle(4)` and re-execute the program. Observe the following differences:

The program executes more slowly.

The overall memory usage is lower and more stable.

This illustrates the difference between *recycling* and *not recycling* worker states – if we do *not* recycle them, Lua must re-create a new worker state every time we create a new thread. This makes the program execute more slowly, but overall the heap usage is *lower* and also *more stable*[18].

This may seem a little counter-intuitive – surely recycling should *save* memory? Well, no – as mentioned above, creating threads is an expensive process in Lua – it involves much more than just allocating memory for the thread. For instance, to create a thread from a string, Lua must invoke the parser to compile the string as part of the creation process, and even when creating a thread from a function, Lua must duplicate not just the function (in case it is changed after the thread begins execution) but also any non-local variables the function references. Retaining the memory used by worker states between each iteration means Lua must allocate *more* memory from the heap to create the threads, whereas if it is *not* recycling it can re-use the memory used by the state during the thread creation process, reducing the need to allocate more memory. The main benefit of recycling is to speed up the thread creation process, not save memory.

[18] Recent improvements in both Lua and in the thread module's memory management have reduced the savings to be had by *not* recycling workers, so unless a program is actually running out of memory, recycling is recommended because it improves program execution speed.

Note that a program can elect to recycle only *some* worker states – it does not need to recycle *all* of them. A program may recycle states of less than the total number of workers if it knows (for instance) that there is a minimum number of workers it would usually need, and extra workers are required only occasionally. In that case, it may be better not to recycle the states for the workers are only required occasionally.

Now we will consider the various memory management techniques used in this program. Some are fairly obvious, some are less so. It is also worth noting that not all of them may be *necessary* in a particular program – but they are all included here to *demonstrate* them:

1. Setting the garbage collection parameters. This is done in the first section of the program:

    ```
    -- set up garbage collection ...
    lua_version = tonumber(string.sub(_VERSION,5,7));
    if lua_version and lua_version < 5.4 then
      -- default mode is incremental, so just set parameters ...
      collectgarbage("setpause", 150);
      collectgarbage("setstepmul", 500);
    elseif lua_version and lua_version < 5.5 then
      -- incremental mode is optional, so set it with parameters ...
      collectgarbage("incremental", 150, 500);
    else
      -- mode and parameter setting now separate in Lua 5.5 ...
      collectgarbage("incremental");
      collectgarbage("param", "pause", 150);
      collectgarbage("param", "stepmul", 500);
    end
    ```

    This process is not as complex as it appears - it is complicated by Lua's changes to the way the parameters to garbage collection have to be specified in various versions of Lua. All that is happening is that the garbage collection is being set to **incremental**, with a **pause parameter** of 150 and a **stepmul parameter** of 500. This increases the "aggressiveness" of the garbage collection process, which can help keep the overall memory usage in check, but at the expense of reduced program execution speed. The best parameters to be used have to be established by trial and error. Consult the Technical Notes at the end of this document, or the Lua documentation for more details.

2. Explicit calls to the garbage collector. The threads module provides a function **gc()** that combines common memory optimization operations, including garbage collection and heap defragmentation - see the description of the function given later in this document.

3. Setting variables to nil. This is done in the thread function:

    ```
    t = nil
    ```

    Since this line is just before the function returns and the memory would presumably be released anyway, it is not obvious why this line helps with memory management. The reason is that explicitly setting a variable to **nil** tells the garbage collector *sooner* than it would otherwise discover for itself that the memory the variable used can be reclaimed. This can make a significant difference, especially if the variable concerned is a large and complex table (which this particular variable is – the variable **t** in this function is a copy of a table that represents the entire **threads** package).

4. Inserting small delays. This is done between program iterations:

   ```
   t.msleep(100)
   ```

   This call allows the threads to finalize and release their allocated memory, and it can also give the garbage collector more opportunities to run.

5. Defragmenting the C heap. This is done explicitly using the **gc** function with a parameter of **1** - i.e:

   ```
   t.gc(1)
   ```

It is worth noting again that not all these techniques are necessary in any particular Lua program (in fact *none* of them are necessary in this small example). However, when memory is tight, one or more of these techniques might help, and it is difficult to predict in advance which may be the most appropriate because it depends heavily on the run-time behaviour of the program.

Example 10 – Using the Propeller module

The next example in this tutorial demonstrates how to add new C functions to Lua. Catalina provides a simple module in the file *demos/catalyst/lua-5.4.8/src/luapropeller.c* which adds a few Propeller-specific functions. Refer to the description of the **propeller** module later in this document to see the functions that have been added, and also examine the source code to see how easily it can be done. In this example, we will just use one of those functions – **togglepin**.

You will find the following program in the file *ex10.lua*:

```
-- define the pins we will use (these are for the
-- P2 EDGE board, and may need changing for other
-- boards) - for example, on the P2 Evaluation
-- board, pins 56, 57 and 58 might be used.
-- Set the pin to -1 to disable it.

pin1 = -1 -- not used
pin2 = 38
pin3 = 39


-- we need 3 workers
threads.workers(3)
-- define a thread function generator:
function PinToggler(pin, msec)
   return function()
      while true do
         propeller.togglepin(pin)
         threads.msleep(msec)
      end
   end
```

```
end

-- start the threads
threads.new(PinToggler(pin1, 1000))
threads.new(PinToggler(pin2, 500))
threads.new(PinToggler(pin3, 250))
threads.wait()
```

Note that, like the **threads** module, the **propeller** module is automatically loaded by Multi-processing Lua, so you don't need to include a **require** statement to use it.

You can execute the program using the following Catalyst command:

```
mlua ex10.lua
```

Observe the LEDs on the P2 Edge board. The program will never terminate. Reset the Propeller to return to the Catalyst prompt.

Here are some replacement lines that might be used for the P2 Evaluation board:

```
-- define the pins to use for the P2 Evaluation board
pin1 = 56
pin2 = 57
pin3 = 58
```

Example 11 – Threads and Coroutines

Even without the threads module, Lua has a simple type of multi-tasking. It has *non-preemptive* multi-tasking in the form of coroutines. The threads module adds *preemptive* multi-tasking. However, within the threads module, each thread is actually implemented as a co-routine, and we can exploit this fact.

For instance, you will find the following program in the file *ex11.lua*:

```
-- Generate a function which can be executed
-- as either a coroutine or as a thread
function Generate(me)
  return function ()
    for i = 1, 10 do
      threads.print(me ..":" .. i .. " ")
      coroutine.yield()
    end
  end
end

-- Execute the functions as coroutines
threads.print("\nFUNCTIONS AS COROUTINES")

-- create 4 functions as coroutines and also
-- wrap the coroutines inside another function
fee = coroutine.wrap(Generate("fee"))
fi  = coroutine.wrap(Generate("fi"))
fo  = coroutine.wrap(Generate("fo"))
fum = coroutine.wrap(Generate("fum"))

-- execute the coroutines
```

```
fee()
fi()
fo()
fum()
-- and again
fee()
fi()
fo()
fum()

-- Execute the functions as threads
threads.print("\nFUNCTIONS AS THREADS")

-- we only need one worker to execute any number of
-- threads if those threads explicitly "yield"
threads.workers(1)

-- create (and start) 4 functions as threads
threads.new(Generate("fee"))
threads.new(Generate("fi"))
threads.new(Generate("fo"))
threads.new(Generate("fum"))

-- wait for all threads to complete
threads.wait()
```

You can execute this program using the Catalyst command:

```
mlua ex11.lua
```

Here is the output of the program:

```
FUNCTIONS AS COROUTINES
fee:1
fi:1
fo:1
fum:1
fee:2
fi:2
fo:2
fum:2

FUNCTIONS AS THREADS
fee:1
fee:2
fee:3
fi:1
fee:4
fi:2
fo:1

...

fee:10
fi:8
```

```
fum:6
fo:7
fi:9
fum:7
fo:8
fi:10
fum:8
fo:9
fum:9
fo:10
fum:10
```

The `Generate` function looks like any of our previous thread generation functions, except it includes a call to `coroutine.yield()`. The **yield** is the key to co-routine behaviour. When a co-routine function is called, and it executes a **yield**, then the function returns control to the caller, but the function itself is not terminated – it is merely *suspended*. When the same function is called again, it does not start from the beginning (as a normal function would), it instead continues execution from where it was last suspended – i.e. just after the last **yield**. This is normal co-routine behaviour (for more details on coroutines, see the Lua documentation).

A Lua thread will be suspended if it explicitly calls `coroutine.yield()`, and also if it calls `threads.put()` or `threads.get()` to send or receive a message and there is no matching thread immediately available to receive a message from, or send a message to.

When we execute this program, the functions are first executed as simple coroutines. Each time one of the functions is called, it executes one iteration and then yields. The program calls each function twice, just to illustrate this fact.

More interesting is the consequence of calling **yield** when the functions are executed as threads instead of co-routines, which is what the program does next. When a *thread* yields then the thread is suspended, but the worker executing that thread is not – it simply moves on to resume execution of the next suspended thread. If there is no other suspended threads it will resume execution of the same thread. Since we started all our functions as threads, this means that one worker (which is essentially one *Posix* or *Catalina* thread) will execute all the functions concurrently (using non preemptive multitasking) *provided those functions are written as co-routines* – i.e. provided they cooperate and **yield** appropriately – and it will do so until the functions actually complete (i.e. not just until they **yield**). If the functions are *not* written as coroutines, then you need multiple workers available to execute them concurrently (using preemptive multitasking). Which of course is very easily done in this example – just change the number of workers from 1 to 4. You can try it, but in this simple example it makes little or no obvious difference to the output.

In summary, with 1 worker the program uses non preemptive multitasking. With 4 workers it uses fully preemptive multitasking. With 2 or 3 workers, it would use a mixture of both.

Because threads are implemented as co-routines we do not always need a separate worker created to execute each thread – if the thread functions can be written as coroutines, only being able to execute a limited number of Lua threads is not necessarily the limitation it may at first seem. This will become clearer in the next example.

However, there is an important limitation to note about exploiting the fact that a thread is implemented as a co-routine – a thread ***cannot hold a mutex locked across a yield***. This is because when a mutex is locked, the mutex actually becomes owned by the underlying worker pthread (i.e. the Posix thread). If the yield changes the worker that is currently being used to execute the thread, then after the yield the mutex will no longer be owned by the thread – so if the thread locks the mutex before the yield, it may not be able to unlock it again afterwards. While it is perfectly possible to use a **mutex** (or a **condition**, which also uses the associated mutex) within a co-routine, the mutex should not be held in a locked state across either an explicit call to **yield**, or an attempt to **put** or **get** a message (which may also perform a **yield** internally). This limitation does not apply to the use of either **rendezvous** or **shared variables**, because all the necessary mutex logic is always performed *between* yields.

Example 12 – More Threads and Coroutines

Example 8 demonstrated that even if we compile Lua as a **COMPACT** program, we only have enough Hub RAM available to have 10 or so Lua functions executing concurrently as *preemptive* Lua threads. But Example 11 showed that we can also make use of Lua's *non-preemptive* multitasking – i.e. co-routines – within a Lua thread. The overheads of co-routines are lower than threads – essentially just some stack space – so can we exploit this capability to be able to execute more than 10 functions concurrently?

As you might expect, the answer is that indeed we can. You will find the following program in the file *ex12.lua*:

```
-- create 4 workers in 4 factories
threads.factories(4)
threads.workers(4)

-- Generate a function which generates 10 co-routines,
-- each of which iterates 10 times. Then execute each
-- of the 10 coroutines 10 times.
function Generate(me)
  return function()
    local f = {}
    for i = 1, 10 do
      -- define the function we want to execute many times
      local my_function = function()
         for j = 1, 10 do
            threads.print(me .. "[" .. i .. "]:" .. j)
            coroutine.yield()
         end
      end
      -- create this function as a co-routine
      f[i]=coroutine.wrap(my_function)
    end
    -- wait till we are told to go
    repeat until threads.shared("go")
```

```
      -- execute each co-routine 10 times
      for n = 1, 10 do
         for m = 1, 10 do
            f[m]()
         end
      end
    end
  end

  -- generate four such functions and execute each as a thread
  threads.new(Generate("A"))
  threads.new(Generate("B"))
  threads.new(Generate("C"))
  threads.new(Generate("D"))

  -- kick things off
  threads.update("go", true)
  -- wait for all threads to complete
  threads.wait()
```

You can execute this program using the Catalyst command:

```
mlua ex12.lua
```

The output of this program will look something like:

```
A[1]:1
A[2]:1
A[3]:1
A[4]:1

...

D[7]:10
D[8]:10
D[9]:10
D[10]:10
```

In the output, **X[n]:m** is the output of the **m**th iteration of the **n**th co-routine – each of which is an instance of `my_function` – that was generated by the call to `Generate("X")`.

If you examine the program and the output, you will see that the program is executing 40 instances of `my_function` concurrently. It does so using a combination of *preemptive* and *non-preemptive* multitasking.

The resulting execution is performed by workers in 4 factories (i.e. using 4 cogs on the Propeller), and so it will execute much faster than the same program would if it were using only coroutines. You can simulate this by changing the number of factories and workers from 4 to 1, which will force the program to use only *non-preemptive* multitasking. If you do so you can see that the program still executes, but much more slowly.

Of course, this is a completely artificial example, and the function being executed (i.e. `my_function`) is almost completely trivial – but it demonstrates that on the Propeller, the threads module really does allow us to exploit all the multi-processing capabilities of the Propeller from Lua.

Example 13 – Using Parallax Locks

The Propeller has built-in support for locks, which can also provide a primitive synchronization mechanism. You will find the following program in the file *ex13.lua.* It is a variation of example 6 that uses the Parallax locks (rather than condition variables) for synchronization:

```
t = require "threads";
p = require "propeller";

-- we want four simultaneous worker threads
t.workers(4);
-- we want to recycle all our workers
t.recycle(4);

-- here is how we will identify the threads, which
-- is also a string for each one to output
eeny  = "eeny ";
meeny = "meeny ";
miney = "miney ";
mo    = "mo\n";

-- a table to hold the lock allocated to each thread, keyed on the string
-- assigned to the thread
lock = {};

-- a function to allocate (and lock) a lock for a thread - the lock is
-- stored in the "lock" table, keyed on the string assigned to the thread
function newlock(me)
  lock[me] = p.locknew();
  if (lock[me] >= 0) then
      if p.lockset(lock[me]) == 0 then
      print(me .. "could not lock " .. lock[me]);
      end
  else
      print(me .. " could not get lock");
  end
  return lock[me];
end
-- allocate a lock for each thread, and store them in the "lock" table
newlock(eeny);
newlock(meeny);
newlock(miney);
newlock(mo);

-- export the "lock" table so all threads have access to the locks
t.export("lock");

-- generate a thread function
function Thread(me, next, count, last)
   return function()
      p = propeller;
      t = threads;
      -- get the lock for this thread from the global "lock" table
      local my_lock = t.shared("lock." .. me);
      -- get the lock for the next thread from the global "lock" table
      local next_lock = t.shared("lock." .. next);
```

```
            repeat
            -- wait for our allocated lock to be released
            while p.lockset(my_lock) == 0 do
            t.msleep(0);
            end
            t.output(me)
            -- release the lock allocated to the next thread
            -- unless we are the last thread in the last round
            if not last or (count > 1) then
            p.lockclr(next_lock);
            end
            count = count-1;
            until count == 0
            t = nil;
         end
      end

      count = 20
      -- create the threads
      t.new(Thread(eeny,  meeny, count, false));
      t.new(Thread(meeny, miney, count, false));
      t.new(Thread(miney, mo, count, false));
      t.new(Thread(mo,  eeny,  count, true));
      -- kick things off
      p.lockclr(lock[eeny]);
      -- wait for threads to complete
      t.wait();
```

Note that you may need to compile this program with **luac** and execute it with **mluax**. If the following Catalyst command does not work:

```
mlua ex13.lua
```

Then try the following instead:

```
luac -o ex13.lux ex13.lua
mluax ex13.lux
```

The output will be identical to that of example 6.

Note the use of the “lock” table to hold the lock number allocated to each worker, the use of **export** to make this table a shared global table, and the use of the **shared** to access the table.

Using locks for synchronization is very efficient, but has significant limitations:

- There are only a limited number of locks, and Catalina needs some of those locks itself, so the number available for use by the program cannot be guaranteed.
- On the Propeller 2, a lock cannot be locked by one cog and then unlocked by another - it must be both locked and unlocked by the same cog - this makes locks suitable for use as a synchronization mechanism only when all threads are executing on the same cog (i.e. in the same factory). This means this example can only ever support one factory.
- The Propeller 2 lock functions (i.e. **locktry** and **lockrel**) allow different threads executing on the same cog to lock and release the same lock - this makes the Propeller 2 style locks unsuitable for use in a multi-threaded program. However, Catalina also implements the Propeller 1 style lock functions (i.e. **lockset** and

lockclr) on the Propeller 2, which do not have that problem - so this program uses those functions instead.

This is the end of the tutorial. We have covered almost all of the functions in the Reference section, but it is worth reading that section and trying out some of the other functions in your own examples. However, we have barely scratched the surface of the Lua language, so it is also worth reading more about Lua.

Also, see the Technical Notes later in this document.

Lua threads Module Reference

Catalina's **threads** module for Lua adds multi-processing capabilities to Lua (for the Propeller 2 only – see below). Multi-Processing Lua is based partly on the Lua *luaproc* module, but that module – which was really a prototype only – has been substantially debugged, modified, extended, and renamed *threads*.

The main differences between the original *luaproc* and the *threads* modules are:

1. Multi-processing has been added in addition to just multi-threading. On the Propeller, this allows Lua to utilise all the available cogs.
2. The basic message-passing object (i.e. the **channel**) has been extended to also act as a mutex and/or a condition variable. Each channel can be used as any one, two, or as all three of these[19]. The functions **mutex** and **condition** are defined as synonyms for **channel**, but it is recommended you use them only if you are EXCLUSIVELY using the channel as a mutex or a condition – otherwise it can get confusing. Also, note that using a condition variable also uses the associated mutex, so (for example) after waiting on a condition, the associated mutex will be locked, and should be explicitly unlocked. Examples are given in the tutorial section of this document, but for a more detailed discussion on condition variables and mutexes, and their interactions, refer to any Posix threads documentation.
3. The global state that all Lua threads can share now allows shared variables to be stored. Functions to export, read and update shared variables are provided. No access control to the shared variables is provided, but this can easily be added at application level using a **mutex**.
4. The threads module is pre-compiled and loaded into the multi-processing version of the Lua interpreter, which is called **mlua** so that it can coexist with the non-threaded version of Lua (i.e. **lua**). This also means it is not usually necessary to use a Lua **require** statement to use the threads module. There is also a corresponding **mluax** which can execute compiled multi-processing lua programs – but note there is no corresponding compiler because the standard Lua compiler (**luac**) can compile all Lua programs whether they use threads or not.

Note that while Lua itself can be compiled for the Propeller 1 if it is equipped with XMM RAM, and Posix threads are supported on both the Propeller 1 and Propeller 2, Multi-processing Lua can be compiled only on the Propeller 2. This is because the Propeller 1 XMM kernel does not support multi-threading, and Multi-processing Lua is too large for the Propeller 1 even if compiled in **COMPACT** mode. On the Propeller 2, Multi-processing Lua can be compiled using any memory model (i.e. **NATIVE**, **COMPACT** or **TINY**).

19 However, note that using a channel for both message passing as well as a mutex or a condition has some limitations – this is addressed in tutorial example 11.

Because the threads module is pre-compiled into Multi-processing Lua, you do not need to `require` it[20]. You just use the module name when referring to the functions, such as:

```
threads.new(code)
```

If you do not want to type `threads` every time, you can define your own synonym for the module, such as:

```
t = threads
t.new(code)
```

Note that this only has a local effect, and so a similar statement needs to be repeated in each context you want to use the synonym (e.g. each chunk of Lua code).

Catalina threads vs Posix threads vs Lua threads

This section gives a brief overview of the key *differences* between the three types of threads.

Catalina threads

Catalina threads are very *light weight* threads. They incur the minimum possible code to implement, and have the minimum possible memory and processor overhead per thread created (about 8 kbytes of initial overhead, and about 150 bytes per thread, not including stack space)[21].

When a particular Catalina thread is actually executing, it executes on the Propeller exactly as fast as the same non-threaded C code would execute – but of course when multiple threads are sharing the same cog, they will each take turns to execute their code.

On a Propeller 2, it is possible to have around 2500 separate Catalina threads executing concurrently.

For an example, see *demos\multithread\test_maximum_threads.c*

The Catalina thread support "primitives" are very simple, and are described in the include files *include\catalina_threads.h* and *include\thread_utilities.h* or the **Catalina Reference Manual**.

Posix threads

Posix threads are *medium weight* threads. They have a higher initial overhead than Catalina threads, and also a higher memory overhead per thread created (about 20k bytes of initial overhead, and about 240 bytes per thread, not including stack space)[22].

However, since Catalina implements Posix threads *using* Catalina threads, their processor overhead is similar.

[20] The `require` Lua statement essentially loads a library module, and unless the module has been pre-loaded, is usually required before you can use any of the functions within the module.

[21] When compiled in COMPACT mode.

[22] When compiled in COMPACT mode.

On a Propeller 2, it is possible to have around 1000 separate Posix threads executing concurrently.

For an example, see *demos\pthreads\test_max_threads.c*

The Posix thread support “primitives” are more sophisticated, and are described in the include file *include\pthread.h* or any Posix documentation.

Lua threads

Lua threads[23] are very *heavy weight* threads, partly because – like Lua itself – they do not execute “natively”. Lua threads must execute in a Lua virtual machine, and there needs to be a separate Lua virtual machine for each concurrent thread.

On a Propeller 2, it is only possible to have a maximum of about 10 to 12 separate Lua threads executing simultaneously, because of their very high overhead (about 180 kb initial overhead, plus about 12-17 kb [24] per thread, not including stack space)[25].

For an example, see tutorial example 8 in this document.

While this may seem to make Lua threads much more limited than Catalina or Posix threads, it is worth pointing out that the execution model used by Lua threads is fundamentally *different* to the execution model used by Catalina threads and Posix threads. With Catalina and Posix threads, to run very many concurrent threads each thread has to be not only identical but also almost trivially simple, and apart from demonstration purposes this is very rarely useful in practice (however, It *can* be done – see tutorial example 12). On a multi-processor system, much less performance improvement is typically gained once you can have one thread executing on each processor, and since Multi-processing Lua can do *that*, it is just a matter of making effective *use* of the capabilities offered. This is addressed further in the next section.

The Lua thread support “primitives” are the main subject of *this* document.

The factory/worker paradigm

The factory/worker paradigm is a common one in multi-processing, where one or more “workers” execute concurrently on one or more “assembly lines” in one or more “factories” to achieve the desired rate of output. Each “factory” is typically dedicated to a single product, but multiple factories can be created if there are enough assembly lines and

[23] Note that Lua uses the term "thread" to mean an instance of an executing co-routine implemented in Lua, as well as an independent thread of execution implemented outside Lua. And of course Catalina and Posix also use the term "thread". The design of the threads module means there is a strong relationship between all these, and in fact a thread module "thread" *is also* a Lua co-routine – this is discussed in tutorial example 11 – but this was a slightly unfortunate choice of terminology on the part of Lua. Since Posix and Catalina already used the term "thread" in its more conventional sense, this was also adopted for the threads module. In this document, the term "Lua thread" (or just "thread") refers to a thread as implemented by the threads module. If any other meaning is intended – such as a coroutine, this is always explicitly stated.

[24] The exact overhead depends on the version of Lua – e.g. it is about 17 kb for Lua 5.1 and about 12 kb for Lua 5.4

[25] When compiled in COMPACT mode.

workers to operate them. Translating this into computer terms, an “assembly line” is typically a single processor (i.e. a "cog" on the Propeller), a “factory” is a set of such processors all dedicated to doing the same job, and each “worker” is a thread of execution currently performing a specific assigned task. And like ideal real-world factories, assembly lines and workers, computer factories can be opened and closed according to demand, assembly lines can be commissioned and decommissioned according to available resources, and (at least in eyes of employers!) workers are interchangeable – they can be moved from one assembly line to another as required, or even from one factory to another.

Those familiar with Catalina’s **parallelizer** functionality may recognize the worker/factory paradigm – it is also adopted by that application. Initially it may look like there are significant differences between the two. For instance, in the parallelizer, all the workers in each factory perform the *same* task, but there can be more than one factory and the workers in each factory typically perform *different* tasks. In the Lua threads case, it initially looks as if each *worker* is performing a *different* task even if they are working in the same factory. But this difference is illusory – each worker in the Lua threads case is actually performing the *same* task – i.e. it is executing a Lua virtual machine. But while each worker is executing exactly the same *code*, it is typically working on different *data* – i.e. the Lua code being executed by the virtual machine.

This also meant that in the case of Lua threads it does not really add anything to be able to assign more than one processor to each factory – all the factories do the same thing anyway. But there are circumstances where we would like to be able to add or remove processors dynamically according to need. While we don’t have the capability to dynamically add or remove processors from factories, we do have the capability to start and stop *factories*, so for Lua threads it makes more sense to just allow multiple *factories* where each factory consists of a single processor (i.e. a cog).

So in the Lua threads module, each *factory* is a cog, and each *worker* is a Posix thread running a Lua virtual machine that can execute a *Lua thread* (i.e. a chunk[26] of Lua code). We can create as many workers and factories as we need (provided there are sufficient resources to do so) to get the job done, but we can only execute as many Lua threads preemptively as we have workers (however, see example 11).

In this document it is also sometimes critical to understand the difference between *Lua threads*, *worker pthreads* and *worker states:*

> *Lua threads* are chunks intended to be executed sequentially, but which may be executed concurrently with other chunks. One Lua thread is created when Lua starts (to execute the main Lua program), and additional Lua threads can be created "on the fly" using the **new** function. Lua threads are destroyed when the

[26] In Lua, the term “chunk” has a specific meaning – it is the unit of execution. A chunk is a sequence of statements which are intended to be executed sequentially. Consequently, it is the natural unit of Lua code to be executed by each worker. In other languages, this might be a function, but in Lua it can also simply be a sequence of Lua statements entered interactively, or encoded in a string.

Lua code being executed terminates (e.g. when a Lua function returns, or the end of a string of Lua statements is reached).

Workers – or *worker pthreads* – are lower level entities (as the name implies, they are in fact Posix threads) that are used to execute Lua threads – but they persist even when the Lua thread terminates, and the same worker pthread may end up executing many different Lua threads at different times. Worker pthreads are created and destroyed using the **workers** function described below.

Finally, *worker states* are Lua states created and destroyed as necessary to enable a worker pthread to do its job. A worker pthread needs a Lua state in which to execute Lua code, but it does not need this state to persist when it is just sitting idle – doing so just consumes resources (i.e. memory) for no benefit if the worker is idle for a length of time.

Without understanding these differences, the **new**, **workers** and **recycle** functions described below would make little sense. The **factory** and **factories** functions, on the other hand, are relatively simple to understand on the Propeller – each factory is simply a cog on which one or more worker pthreads may execute a chunk of Lua code. They are what allow Multi-processing Lua to execute Lua threads in true parallel, and allow Lua to exploit all the capabilities of the Propeller.

Process management functions

The following sections describe each of the functions offered by the Lua threads module.

new

Syntax: `new(<func> or "code")`

Start a new Lua thread from a function, or from a string containing Lua code. Note that starting a Lua thread does not start a worker to *execute* that thread – if there is no worker available, the Lua thread waits until one becomes available. So we always have to ensure that there are sufficient workers available, which can be done using the **workers** function. Also note that the threads module always begins with zero workers – if none have been created via the **workers** function, a call to the **new** function will never return.

`newproc` is a synonym for `new`.

wait

Syntax: `wait([<condition>])`

With no parameter, waits for all Lua threads to complete. Note that it only makes sense to use this function in the top-level or main Lua thread – if you use it in another thread, that thread will never complete!

The <condition> parameter can be a string, or a string variable.

With a parameter, waits for the specified condition (i.e. `wait` with a parameter is a synonym for `wait_for`).

When used to wait for Lua threads to complete, the wait function does a garbage collection and also defragments the C heap before returning – this speeds up the process of reclaiming the memory used by the threads that have just completed.

workers

Syntax: `<num> = workers([<num>])`

Get or set the number of worker pthreads.

With no parameter, returns the current number of worker pthreads.

With a numeric parameter, requests that worker pthreads be created or destroyed to suit the new number (note the worker pthreads must be idle to be destroyed – if they are not, they will not be destroyed until the Lua thread they are currently executing terminates). Always returns the current number of worker pthreads (note that this may not be the same as the number requested in the call, since workers are not actually created or destroyed by this function and this may not happen immediately – the returned value may be anywhere between the current number and the requested number of workers).

Note that there must be enough worker pthreads available to run all the Lua threads that must execute concurrently, or the program will wait until a worker pthread becomes available (e.g. a currently executing Lua thread terminates). If none of the currently executing Lua threads terminates, the program will never complete.

Also note that (unlike the **luaproc** module) the **threads** module always begins with zero workers – if none are created, no Lua threads will execute and the **new** function will never return.

`setnumworkers` is a synonym for `workers` with a parameter specified.

`getnumworkers` is a synonym for `workers` with no parameter specified.

factories

Syntax: `<num> = factories([<num>])`

Get or set the number of factories. On Catalina, this function specifies the maximum number of cogs that will be used to run worker pthreads – whether this many factories are started or not depends on how many free cogs are actually available. In any case, the number actually available is returned, so to specify that all available cogs be used, simply request 8 factories.

The default value is 1 factory, and the number cannot be reduced below this. With no parameter, this function just returns the number of factories currently available (which will always be at least 1).

On other platforms, this function may do nothing and always just return 1.

The number of factories can be increased at any time, but should only be reduced when there are no worker pthreads executing, since it must terminate and restart all worker pthreads. This means the number of factories should only be reduced by the main thread, when that thread is executing in factory 1.

Typically, this function will only be called as one of the first thread functions used by a Multi-processing Lua program.

Newly created worker pthreads will be distributed among all the available factories, and so there should generally be at least as many workers as there are factories (or else having the extra factories is just a waste of resources) - but even so if workers are starting and stopping essentially at random then it can happen that all the workers are executing in one factory while the other factories are idle, so there is a facility provided to allow workers to move themselves between factories (see the **factory** function).

factory

Syntax: `<num> = factory(<num>)`

In Catalina, specifying a number moves the worker to that factory if possible. Note that the factory number is not necessarily the cog number – if "n" factories were started via the **factories** function, then they will always be numbered from 1 to n regardless of which cogs are actually being used.

This function returns the factory cog on which the Lua thread that called it is currently executing (which will be the new factory if called with a parameter and the call succeeds) or zero if it fails.

On other platforms, this function may simply always return 1.

Note that the main Lua thread cannot be moved to another factory unless there is at least one other worker executing on factory 1 – the easiest way to ensure this is to create at least as many workers as there are factories.

For example:

```
threads.factories(5)    -- create 5 factories
threads.workers(5)      -- one worker will end up in each factory
```

recycle

Syntax: `recycle(<num>)`

Set the number of worker *states* (not worker *pthreads*) to be *recycled* between uses (zero means no recycling – i.e. worker state will be deleted when the Lua thread terminates and a new worker state will be created whenever a new Lua thread is created.

Recycling worker states can save the overhead of creating a new one each time a new Lua thread is created, but since the states remain in existence even when they are not being used it can result in higher overall memory usage.

stacksize

Syntax: `<num> = stacksize([<size>])`

If a numeric parameter is specified, set the size (in bytes) to be used as the stack size for all new workers (only affects new workers, so this should appear either before any worker pthreads are created, or after setting the number of worker pthreads to zero).

Always returns the current stack size.

Note that it is *possible* to allocate each thread a different stack size, but not recommended. It is only possible if the worker pthreads and Lua threads are carefully created in a specific order, with judicious calls to the **stacksize**, **workers** and **new** functions in a specific order, and only if the threads involved never terminate.

Common Channel/Mutex/Condition functions

channel

Syntax: `channel(<name>)`

Create a new named communications channel/mutex/condition.

The <name> parameter can be a string, or a string variable.

`newchannel` is a synonym for `channel`

`mutex` is a synonym for `channel`

`condition` is a synonym for `channel`

destroy

Syntax: `destroy(<name>)`

Destroy a new named communications channel/mutex/condition.

The <name> parameter can be a string, or a string variable.

Any Lua threads waiting on the channel or the condition will be woken (but note they should check the condition they were waiting on is true!).

`delchannel` is a synonym for `destroy`

Message-passing functions

put

Syntax: `put(<name>, <tuple>)`

Blocks until there is a receiver, or the channel is destroyed. If there is no receiver waiting. the thread will do a co-routine **yield**.

The <name> parameter can be a string, or a string variable.

<tuple> = one or more values to send.

`send` is a synonym for `put`

aput

Syntax: `aput(<name>, <tuple>)`

Returns nil immediately if there is no receiver waiting.

The <name> parameter can be a string, or a string variable.

Always returns immediately if the named channel does not exist.

<tuple> = one or more values to send.

`send_async` is a synonym for `aput`

get

Syntax: `<tuple> = get(<name> [, async ])`

If the async boolean flag is false or not present, blocks until there is a sender or the channel is destroyed.

If async is false, and there is no receiver waiting, the thread will do a co-routine **yield**.

If async is true, returns nil immediately if there is no sender waiting.

The <name> parameter can be a string, or a string variable.

Always returns immediately if the named channel does not exist.

`receive` is a synonym for `get`

aget

Syntax: `<tuple> = aget(<name>)`

Same as get, but with the async flag specified automatically.

The <name> parameter can be a string, or a string variable.

`receive_async` is a synonym for `aget`

Mutex functions

lock

Syntax: `lock(<name>)`

Lock the mutex associated with the named channel.

The <name> parameter can be a string, or a string variable.

Note that this is a NEW mutex, not the existing channel mutex!

`true` is returned WHEN the lock succeeds or nil and an error message on any error.

Note that while it is fine for a co-routine to use mutexes, they should not be held locked by the co-routine when it performs an explicit co-routine **yield**, or a threads **put** or a **get** (which may do a yield internally).

trylock

Syntax: `trylock(<name>)`

Try and lock the mutex associated with the named channel.

The <name> parameter can be a string, or a string variable.

Note that this is a NEW mutex, not the existing channel mutex!

`true` is returned if the lock succeeds, nil if not or nil and an error message on any error.

Note that while it is fine for a co-routine to use mutexes, they should not be held locked by the co-routine when it performs an explicit co-routine **yield**, or a threads **put** or a **get** (which may do a yield internally).

unlock

Syntax: `unlock(<name>)`

Unlock the mutex associated with the named channel

The <name> parameter can be a string, or a string variable.

`true` is returned if the unlock succeeds or nil and an error message on any error.

Condition functions

wait_for

Syntax: `wait_for(<name>)`

Waits for the condition to be signalled (or for the named channel to be destroyed)

The <name> parameter can be a string, or a string variable.

`true` is returned if the condition occurs or nil and an error message on any error.

`wait` is a synonym for `wait_for` if a parameter is specified

Note that while it is fine for a co-routine to use conditions, the associated mutex should not be held locked by the co-routine when it performs an explicit co-routine **yield**, or a threads **put** or a **get** (which may do a yield internally).

signal

Syntax: `signal(<name>)`

Signal a condition, releasing a single Lua thread waiting on the condition

The <name> parameter can be a string, or a string variable.

`true` is returned if the signal is sent or nil and an error message on any error

broadcast

Syntax: `broadcast(<name>)`

Signal a condition, release all Lua threads waiting on the condition.

The <name> parameter can be a string, or a string variable.

`true` is returned if the broadcast is sent or nil and an error message on any error

rendezvous

Syntax: `rendezvous(<name>)`

Call a rendezvous. A rendezvous is a convenience function that implements the following logic in a single call:

```
lock(condition)
broadcast(condition)
wait_for(condition)
unlock(condition)
broadcast(condition)
```

The result of this logic is that when *one* thread calls a rendezvous, it will be suspended until *another* thread calls the same rendezvous – then *both* threads will proceed. The most important feature of a rendezvous is that it is symmetrical – e.g. unlike message passing it does not require one thread to be a sender and the other thread to be a receiver – and that it therefore does not matter in which order the threads call it.

The <name> parameter can be a string, or a string variable.

`true` is returned if the rendezvous succeeds, or nil and an error message on any error

Global/Shared Variable functions

shared

Syntax: `shared(<name>)`

Read the named variable (which can be a table) from the shared global state – e.g. "a" or "a.b.c" - or nil if the named variable does not exist.

The <name> parameter can be a string, or a string variable.

Note that for tables, only one level of table depth is returned, and only table keys with simple values (nil, boolean, number or string) will be returned. To retrieve more depth, retrieve each sub table separately.

For example, if table "a" has a table value for key "b":

```
a = threads.shared("a")
a.b = threads.shared("a.b")
```

update

Syntax: `update(<name>, <value>)`

Update the named variable in the shared global state – e.g. "a" or "a.b.c".

The <name> parameter can be a string, or a string variable.

The <value> must be a simple value (i.e. nil, boolean, number or string).

export

Syntax: `export(<name>)`

Export the named variable (which can be a table) from the Lua threads current environment to the shared global state (to export everything, export "_G" – an explanation of this is beyond the scope of this document – see the Lua reference manual for more details on "_G").

The <name> parameter can be a string, or a string variable.

Note that for tables, only one level of table depth is exported, and only table keys with simple values (nil, boolean, number or string) will be exported. To export more depth, export each sub-table separately.

For example, if table "a" has a table value for key "b":

```
threads.export("a")
threads.export("a.b")
```

The export function is often used to export the global environment of the main Lua thread after it has set it up for other Lua threads to use.

Miscellaneous functions

sleep

Syntax: `sleep([<secs>])`

Sleep for the specified number of seconds.

If the parameter is zero or not specified, sleep just does a Posix thread yield. This is useful if a thread has nothing to do as it allows other threads – and also the Lua garbage collector – to execute.

msleep

Syntax: `msleep([<msec>])`

Sleep for the specified number of milliseconds.

If the parameter is zero or not specified, msleep just does a Posix thread yield. This is useful if a thread has nothing to do as it allows other threads – and also the Lua garbage collector – to execute.

print

Syntax: `print(<tuple>)`

Print each element of the tuple protected by a lock to prevent interleaving the output of different Lua threads. Each element will be separated by a tab, and the whole set of tuples will be terminated by a newline.

This is designed to simulate Lua's own **print** function, but safer for competing Lua threads. In particular, a single string is guaranteed to be output intact even if multiple Lua threads are printing at the same time (provided they all use the thread print or output functions). Therefore, it may be useful to concatenate all the output into a single string rather than specify multiple parameters.

output

Syntax: `output(<tuple>)`

Print each element of the tuple unformatted, but protected by a lock to prevent interleaving the output of different Lua threads. In particular, a single string is guaranteed to be output intact even if multiple threads are printing at the same time (provided they all use the thread print or output functions). Therefore, it may be useful to concatenate all the output into a single string rather than specify multiple parameters.

sbrk

Syntax: `<num> = sbrk([<defrag>])`

Return the current value of the C system break (i.e. the current top of the heap), and if the defrag flag is `true`, defragment the heap memory. Mainly used for debugging, to see how much heap memory Lua is using, and/or spot memory leaks or identify programs that fail because they run out of memory. See also the Technical Notes later in this document.

gc

Syntax: `<num> = gc([<optimize>])`

Perform a "thread safe" garbage collection, and optionally perform some additional memory optimization functions. With no parameter specified, the function just performs a garbage collection. If the optional integer parameter is specified, then its value means to also:

0	return the current value of **sbrk**.
1	defragment the heap memory and return the current value of **sbrk**.
2	free any internal code buffer space allocated during thread initialization. See also the Technical Notes later in this document. Returns **0**.

Note that to ensure that *all* garbage collections performed in the program are thread-safe, *only* the **gc()** function should be used to collect garbage - normal Lua background garbage collection should be turned off via **collectgarbage("stop")**

version

Syntax: `<num> = version([<what>])`

Returns a version number depending on the optional parameter, which must be a string. The parameter can be:

"lua" (or no parameter) - returns the LUA_VERSION_NUM associated with the Lua version. The main use for this is to allow for incompatibilities between Lua versions, such as the differences in the garbage collection options in various Lua releases. The possible return values are:

501 – Lua Version 5.1.x

502 – Lua version 5.3.x

503 – Lua version 5.3.x

504 – Lua version 5.4.x

"hardware" - returns the Propeller hardware version. This can be::

1 - Propeller 1

2 - Propeller 2

0 - not a Propeller (e.g. Windows or Linux)

"module" (or any other string) - returns the MODULE_VERSION_NUM associated with the module. For example:

611 - Threads module version 6.1.1

Synonyms for threads functions

Several of the threads functions have synonyms defined:

synonym	threads function	additional notes
`newproc`	`new`	
`receive`	`get`	
`send`	`put`	
`receive_async`	`aget`	
`send_async`	`aput`	
`setnumworkers`	`workers`	(with a parameter specified)
`getnumworkers`	`workers`	(with no parameter specified)
`newchannel`	`channel`	
`mutex`	`channel`	

synonym	threads function	additional notes
`condition`	`channel`	
`delchannel`	`destroy`	
`wait`	`wait_for`	(with a parameter specified)

These synonyms are useful for emulating the original luaproc module, but note the following:

It still may be necessary to either modify **luaproc** to **threads** in the source, or else include a line like `luaproc = threads` in each new Lua context (e.g. in the code of each new Lua thread).

With no parameter, **`wait`** acts like the luaproc **wait** function, which waits for all Lua threads to complete, but with a parameter it is a synonym for **wait_for**.

With no parameter, **`setnumworkers`** acts like **`getnumworkers`**, but also does a heap defragment (which is different from the Lua garbage collector – see the Technical Note on memory management later in this document).

In **`setnumworkers`**, the number of workers can now be set to zero to stop all worker pthreads (e.g. if you want to change the stack size, first stop all worker pthreads, otherwise some workers will have the old stack size, and some will have the new stack size, which can make things very confusing as the same Lua program may work sometimes but not others, depending on which worker pthread it is using!).

The **put** and **get** functions (aka **`send`** and **`receive`**) now handle tables (only one level of depth is supported, and only keys with simple values – i.e. nil, boolean, numbers and strings) will be sent or received.

Lua propeller Module Reference

The Lua **propeller** module is included as an example of adding Propeller-specific functions to Lua. It is implemented in the *demos/catalyst/lua-5.4.8/src/lprop.c C* source file.

The **propeller** module is pre-compiled into Catalina's version of both Lua and Multi-processing Lua. This means you do not need to `require` it. Just use the module name when referring to the functions, such as:

```
propeller.setpin(pin, value)
```

See the description of **setpin** below. If you do not want to type `propeller` every time, you can define your own synonym for the module, such as:

```
p = propeller
```

Currently, the functions described in the sections below are implemented, which corresponds to the functions of the same name implemented in the C libraries[27]. They take and return the same values as the corresponding C functions.

The **propeller** module is also supported in the Propeller 1 version of Lua, with the exceptions noted in this section.

Pin functions

Lua analogues of the pin I/O functions:

`setpin(pin, value)`	sets pin to output and sets value
`getpin(pin)`	sets pin to input and gets value
`togglepin(pin)`	sets pin to output and toggles value

File Processing functions

Lua analogues of the DOSFS wildcard/globbing functions:

```
mount()

scan(function, directory, filename)
```

The file system must be mounted before being scanned. Mounting the DOSFS file system does not interfere with the normal Lua file processing functions.

The scan function accepts a directory to scan and also the filenames to scan for, which can include wildcards. When a directory is scanned, the specified function is called on each matching file name. The function should be one that accepts the file name, DOSFS file attribute, and the file size.

[27] Apart from the **execute** function, which has no direct C analogue, since it relies on Catalyst and not just the C runtime. However, this function is trivial to implement In C, since all it does is create a file and then reboot the Propeller.

Here is a small but complete complete example:

```
-- print the file name only
function print_entry(name, attr, size)
   print("   " .. name)
end

-- mount the DOSFS volume
propeller.mount()

-- scan the DOSFS directory specified in arg[1] and call print_entry
-- on any files matching the wildcard pattern specified in arg[2]
propeller.scan(print_entry, arg[1], arg[2])
```

See the Lua demo program *list.lua* for a more sophisticated example of using these functions.

Execute function

A Lua function to execute any Catalyst command from within Lua (note that doing this reboots the Propeller):

```
execute(command, filename)
```

For example:

```
propeller.execute("list *.bas")
```

The first parameter is the command (which may include parameters), and the second (optional) parameter is the file name to write this command to. This defaults to "EXECONCE.TXT", which means the command will only be executed once, but it can be used to write the command to any file. For instance:

```
propeller.execute("lua", "AUTOEXEC.TXT")
```

will cause the Propeller to execute **lua** on each reboot. To disable this from within Lua, just delete the file by executing a Lua command like:

```
os.remove("AUTOEXEC.TXT")
```

Lock functions

Lua analogues of Propeller 1 and Propeller 2 style lock functions:

`locknew()`	allocate a lock, return lock or -1 if none available
`lockret(lock)`	return a lock to the lock pool
`lockset(lock)`	try to set a lock (P1 style), return 1 on success or 0 on failure
`lockclr(lock)`	release a lock (P1 style)
`locktry(lock)`	try to set a lock (P2 style), return 1 on success or 0 on failure

`lockrel(lock)`	release a lock (P2 style)

Hub Functions

Lua analogues of various Hub functions:

`cogid()`	return the number of the current cog
`clkfreq()`	return the current clock frequency
`clkmode()`	return the current clock mode - the result is P1 or P2 dependent
`getcnt()`	return the current system counter, either as one integer (P1) or as 2 integers (lower 32 bits, upper 32 bits) (P2)
`muldiv64(m1, m2, d)`	return the 32 bit value of the 64 bit calculation (m1*m2/d)

Other functions

`version(what)`

Returns a version number - accepts the same optional parameters and returns the same values as the corresponding function in the threads module.

`setenv(name, value, overrwrite)`

Sets an environment variable. Supported only on the Propeller 2. The **name** and **value** are string values, and **overwrite** is an integer - **0** means do not overwrite the value if the variable already exists, **1** means overwrite an existing variable with the new value.

Note that the environment variable is not added to the *current* environment - it is added to the environment that will be used by *subsequent* instances of Lua.

Note also that the corresponding **getenv** function is not in the **propeller** module - it is a standard Lua function that already exists in the **os** module.

`unsetenv(name)`

Unsets (deletes) an environment variable. Supported only on the Propeller 2.

Note that the environment variable is not deleted from the *current* environment - it is deleted from the environment that will be used by *subsequent* instances of Lua.

`sleep(seconds)`

Sleep for the specified number of seconds.

`msleep(seconds)`

Sleep for the specified number of microseconds.

`sbrk(defrag)`

Similar to the same function in the threads module.

Lua HMI Module Reference

The Lua **hmi** module is included to add Propeller-specific HMI functions to Lua. It is implemented in the *demos/catalyst/lua-5.4.8/src/lhmi.c C* source file.

HMI functions

Lua analogues of the keyboard HMI functions:

```
k_get, k_wait, k_new, k_ready, k_clear
```

Lua analogues of the screen HMI functions:

```
t_geometry, t_mode, t_setpos, t_getpos, t_scroll, t_color,
t_color_fg, t_color_bg, t_string, t_char
```

Lua analogues of the mouse HMI functions:

```
m_button, m_abs_x, m_abs_y, m_delta_x, m_delta_y, m_reset,
m_bound_limits, m_bound_scales, m_bound_x, m_bound_y
```

Each of these HMI functions accepts the same parameters and returns the same values as their C counterparts. See the *Catalina Reference* Manual for details.

Note that to save space, the mouse functions will not be included if the Catalina symbol **NO_MOUSE** is defined when Lua is compiled (which it will be if you use the **build_all** scripts to build Lua – to change this you can edit **Makefile.Catalina** to remove the **-C NO_MOUSE** option). You can detect whether mouse functions have been included from within Lua itself by testing if any of them are nil, such as:

```
if (hmi.m_button) then
   -- mouse functions have been included
else
   -- no mouse functions
end
```

Other functions

```
version(what)
```

Returns a version number - accepts the same optional parameters and returns the same values as the corresponding function in the threads and propeller modules.

Technical Notes

Lua memory management

Lua does automatic memory management. Programs can *create* objects (tables, strings, threads etc), but there is no function to *delete* objects. Instead, Lua automatically deletes objects that become garbage, using a *garbage collector* that runs periodically.

This is very handy, but dynamic memory management can be a problem in highly memory constrained environments. In particular, Lua programs that use a lot of strings and do string manipulations can end up with extremely fragmented memory, and even quite small programs can run out of memory unexpectedly. An example of this is shown in the tutorial example 9, which enables more aggressive garbage collection in order to avoid running out of memory.

There are several aspects to this issue, and all may have to be considered when writing Lua programs:

Lua thread stack size

It is worth considering the stack space needed by Lua threads. Only the main Lua thread has unconstrained stack space – all other Lua threads have strictly limited stack space which is allocated on thread creation. The default is 4000 bytes (i.e. 1000 longs), which is sufficient for many programs (e.g. it is sufficient for most of the tutorial examples). The necessary stack size is difficult to determine in advance, but allocating more stack space (using the **stacksize** function) should be tried if the program does not behave as expected – but it is also worth spending some time making the stack as *small* as possible. Finally, remember that all workers should have the same stack size, so it should normally be specified before any workers are created, unless you know in advance that a specific worker will only ever be used to execute a specific Lua thread.

Thread-safe memory management and C heap fragmentation

The C heap can become fragmented when many small **malloc**, and **free** statements are executed, and Lua is particularly prone to doing this! The C memory management functions try to avoid the need to do any explicit memory defragmentation, but it can still help in cases when *many* small blocks have been allocated and then freed – in such cases it can become impossible to allocate a large block even when there is sufficient heap space available, because it has been broken up into multiple small blocks. Defragmenting the heap recombines multiple small blocks of memory back into larger blocks if it discovers they are contiguous. However this process can take time, so it should only be performed if absolutely necessary.

Catalina offers a C function called **malloc_defragment()** which can be called to do this. From Lua, this function is called whenever the threads module functions **sbrk** or **gc** are called with a **true** parameter (i.e. any parameter other than **false** or **nil**). The **gc** function is thread-safe, but as a result it is also slightly more expensive to invoke than **sbrk**. However, it is the one that should be used when explicit garbage collection is needed *within* a thread function. If the **gc** function is called with a **true** parameter, it also does deallocation of internal buffers that were allocated automatically during thread creation, and are never deallocated. This is done for performance reasons, since Lua does not know when the thread creation process is complete. It means that to maximize memory, **gc** should (if possible) be called with a **true** parameter by the main program only *after* all threads have been created, but not *while* threads are still being created (since de-allocating and then re-allocating the required internal buffers can actually be counter-productive in terms of memory usage).

For an example of the **sbrk** function, see tutorial example 8. For an example of the **gc** function, see tutorial example 9.

Lua memory fragmentation

Lua normally manages its own memory automatically, but the Lua garbage collector can be manually invoked to either perform explicit garbage collection, or to tweak the garbage collection parameters.

Appropriate calls to Lua's **collectgarbage** function can make a program work that would otherwise run out of memory.

Here are some of the garbage collector options[28]:

`collectgarbage()`	perform a garbage collection immediately.
`collectgarbage("collect")`	same as above.
`collectgarbage("count")`	return the current memory usage in kilobytes.
`collectgarbage("step", <num>)`	perform some incremental steps of garbage collection, rather than a full collection cycle.
`collectgarbage("setpause", <num>)`	set the interval between collections. A value of 100 is normal. The maximum value is 1000. Larger values reduce the aggressiveness of garbage collections. A value of 100 or lower means there is no pause between collections.
`collectgarbage("setstepmul", <num>)`	set the amount of work performed on each collection. A value of 100 is normal. The maximum value is 1000. Larger values increase the aggressiveness of the garbage collector, but also increase the processor overhead. A value of lower than

[28] Note that this section mentions only some of the **collectgarbage** options, and also only those that are common to both Lua 5.1 through Lua 5.4 (although some are deprecated in 5.4 and are likely to be removed in subsequent versions). Lua 5.4 also has *additional* **collectgarbage** options, and Lua 5.5 has slighlty *different* options to **collectgarbage** - check the Lua reference manual for the specific version of Lua in use.

100 may prevent the garbage collector from ever completing and should not be used.

Garbage collection can be an expensive process and it is better to use the garbage collector as sparingly as possible. However, Lua's garbage collector is *incremental*, which means it does not need to do all its work in a single step – this is the purpose of several of the parameters, which specify how much work will be performed in each incremental step. First, try adjusting the parameters to be slightly more aggressive and see if this is sufficient, or just just doing a few incremental steps at key points in the program (e.g. after starting or stopping a thread). The values for the parameters have to be established by trial and error. Invoking the garbage collector to do a complete collection should be avoided if possible.

Note that it is typical for Lua to allocate a lot of memory from the C heap initially, and then less and less as Lua finds it can re-use the memory it reclaims from garbage collection. This makes the **threads** module's **sbrk** or **gc** functions are very useful diagnostic tools – printing the returned value periodically will tell you whether the Lua memory allocation has stabilised, or if the program has a continuous memory leak (which can happen even with automatic memory management if variables are left around with values assigned – once a variable's value is no longer required, assign the variable a **nil** value to tell Lua it can collect the unused values. This is particularly important with string values, which can occupy a lot of memory, and when string operations are used because these can create many temporary string values, consuming an unexpectedly large amount of memory until the next garbage collection).

For an example of using garbage collection, see tutorial example 9.

Lua versions supported

The **threads** module has been tested with the following versions of Lua:

Lua 5.1.5

This version of Lua has the smallest initial memory footprint, and used to be the default version. It can be easily downloaded and modified to incorporate the **threads** and **propeller** modules.

Lua 5.3.6

This version of Lua added integers (Lua 5.1.5 has only floats) which makes it better suited to the types of application for which the Propeller is typically used. However, on balance, its larger footprint makes it less attractive than version 5.1.5. It can be easily downloaded and modified to incorporate the **threads** and **propeller** modules.

Lua 5.4.4

This version of Lua improved Lua's memory management. While it has a larger initial footprint, the improved memory management ameliorates this disadvantage to a large extent, and the overall memory usage can actually be lower than version 5.1.5 if the Lua program uses 8 or more worker pthreads.

Lua 5.4.8

This is now the default version of Lua fully integrated with Catalina – e.g. it is the version built by the Catalyst **build_all** scripts.

Lua 5.5.0

This version is now included with Catalina as a stand-alone program – e.g. it is not yet the version fully integrated into Catalina, or built by the Catalyst **build_all** scripts.

Multi-processing Lua Source Files

Multi-processing Lua makes no changes to Lua itself[29]. All the additions to Lua to implement the **threads** and **propeller** modules, and to make them work with Catalina 5.0, are in the following C source files:

`luaconf.h`	Various configuration parameters are set in this file, such as telling Lua to use ANSI C and (in some versions of Lua) the size of integers and floats.
`linit.c`	The only change to this file is to add the **propeller** and **threads** modules as one of the base modules that will be opened automatically. The threads module is only included if the program is compiled to use Catalina's threads library.
`luathreads.h`	Defines the threads module.
`luathreads.c`	Implements the threads module.
`luathreadsched.h`	Defines the threads module scheduler.
`luathreadshed.c`	Implements the threads module scheduler.
`luapropeller.c`	Implements the propeller module.
`Makefile`	Modified to know about Multi-processing Lua, so **mlua** and **mluax** can (for example) be compiled for Windows or Linux. For Catalina, use `Makefile.Catalina` instead (this is done automatically by the build_all scripts in the Catalyst and Lua folders).
`Makefile.Catalina`	Makefile for Catalina's version of Lua and Multi-processing Lua. Eventually, this should be rolled into the `Makefile`.

[29] There is currently one Lua file – `lauxlib.c` – that has been modified to initialize all newly allocated memory blocks to contain all zeroes. This is required because Lua apparently depends on the assumption that all allocated memory is initialized, which may be true on some platforms but is not actually guaranteed by C, and is not true for Catalina on the Propeller. When this dependency is found, this change can be removed.

Using the Lua threads module with other C compilers

The Lua threads module is *not* Catalina-specific. Like Lua, it is written entirely in ANSI C and can be compiled and used with any compiler that supports ANSI C and Posix threads.

An example of doing this is provided in the Lua 5.1.5 **Makefile** (Catalina currently has its own Makefile, called **Makefile.Catalina**). This Makefile knows how to compile Multi-processing Lua as well as normal Lua, on multiple platforms.

For example, on Windows, you can make Multi-processing Lua using a command like:

```
make generic all
```

or

```
make mingw all
```

However, note that some functions (such as the ability to create multiple factories) will only work “out of the box” on Catalina. On other platforms only one factory will be supported unless the code is modified, because the code to do this is platform-specific.