Tutorial: Lightweight Processes

Due to its Erlang foundation, an LFE program is composed of anywhere from 1 to hundreds of thousands of lightweight processes. In fact, a 2005 message to comp.lang.functional reported spawning 20 million messages in a ring benchmark (on a 1.5 GHz SPARC with 16 GB RAM).

This is possible in part because each process operates independently with its own private memory, communicating via message passing, and the overhead for an Erlang process is pretty low (~1.5k for 32-bit and ~2.7k for 64-bit; see the Efficiency Guide's section on processes for more info).

This tutorial aims to decrease the mystery around Erlang processes and how to use them in LFE programs.

1 Interacting with Processes

1.1 Dump and `flush`

Processes in LFE are built from functions. These running functions communicate with other running functions via messages. When you start up the LFE REPL, you're using an Erlang process, and you can communicate with it just like any other process.

Let's get the REPL's process id and then send messages to the REPL using the PID:

```
> (set pid (self))
<0.30.0>
> (! pid '"Testing: 1, 2, 3!")
"Testing: 1, 2, 3!"
> (! pid '"This is another test message...")
"This is another test message..."
> (! pid (list 1 2 3))
(1 2 3)
>
```

The messages are sitting in the inbox of the process they were sent to, the REPL. If we flush the REPL's inbox, we can see them:

```
> (: c flush)
Shell got "Test1"
Shell got "Testing: 1, 2, 3!"
Shell got "This is another test message..."
Shell got [1,2,3]
ok
>
```

1.2 Getting Classy with `receive`

As you might imagine, there's a better way to do this. Let's send another message to the REPL's message queue (inbox):

```
> (! pid (list 1 2 3))
(1 2 3)
>
```

Now let's take a look at that message without flushing the queue:

```
> (receive
    ((list a b c)
     (: io format '"results: ~p ~p ~p~n" (list a b c))))
results: 1 2 3
ok
>
```

If there is a message in the inbox matching the pattern we have defined (in this case, a list of length 3), then we will have access to the data that is matched and bound to the variables. For more information on pattern matching, see the tutorial.

If there are a bunch of messages in the inbox, they will all be iterated over until a match is found:

```
> (set pid (self))
<0.30.0>
> (! pid (tuple 1 2 3))
#(1 2 3)
> (! pid (tuple 2 3))
#(2 3)
> (! pid (tuple 3))
#(3)
> (receive
    ((tuple a)
     (: io format '"results: ~p ~n" (list a))))
results: 3
ok
```

Let's confirm that only the last message we entered was matched:

```
> (: c flush)
Shell got {1,2,3}
Shell got {2,3}
ok
>
```

1.3 Shell `spawn`

So far, we've only look at the process for the REPL itself. We'd like to expand our horizons and look at creating a process in the REPL, writing to it instead of our shell.

However, we are faced with some difficulties:

- LFE doesn't let us define functions (or macros) in the REPL, and
- Erlang's `spawn` function takes a module and function as a parameter.

We can sort of get around that first point using `lambda`:

```
> (set print-result
    (lambda (msg)
```

```
      (: io format '"Received message: '~s'~n" (list msg))))
#Fun<lfe_eval.10.53503600>
> (funcall print-result '"Zaphod was here.")
Received message: 'Zaphod was here.'
ok
>
```

Let's update this function so that it can respond to messages when it's running as an Erlang process using the call to `receive`:

```
> (set print-result
    (lambda ()
      (receive
        (msg
          (: io format '"Received message: '~s'~n" (list msg))))))
#Fun<lfe_eval.21.53503600>
>
```

Now that we've got our message-capable function, let's `spawn` it and capture the process id so that we can write to it:

```
> (set pid (spawn (lambda () (funcall print-result))))
<0.66.0>
> (! pid '"Time is an illusion. Lunchtime doubly so.")
"Time is an illusion. Lunchtime doubly so."
Received message: 'Time is an illusion. Lunchtime doubly so.'
>
```

As you can see, when we send our message to the process we started with `spawn`, the process' function prints out the message that it received from us.

We had to go through some gymnastics here, due to the limitations of the shell and using `funcall` in a `spawn` call.

Up next: in an anti-intuitive twist, you'll see that doing the same thing from a module is more clear that doing it in the shell ;-)

2 Processes in Modules

2.1 Shell `spawn`: The Sequel

In the last section, we were all primed to explore spawning processes from the REPL. As we explored, we discovered that message passing in the REPL is a little cumbersome. You were also promised that it would be cleaner when we moved the code to modules. Let's see if that's true …

Save the code below to `messenger.lfe`:

```
(defmodule messenger
 (export (print-result 0)))

(defun print-result ()
  (receive
    (msg
```

```
      (: io format '"Received message: '~s'~n" (list msg)))))
```

Then start up `lfe`, compile your new module, and spawn our print function:

```
> (c '"messenger")
#(module messenger)
> (set pid (spawn 'messenger 'print-result ()))
<0.51.0>
```

Great! It works as expected. Now let's play… by sending it a message from the REPL:

```
> (! pid '"Zaphod was here.")
"Zaphod was here."
Received message: 'Zaphod was here.'
```

The only problem with our solution is that it's a one-shot deal; subsequent sends to the pid won't call our function, since that function is no longer running. We can change that, though: let's make sure that once it prints the message, it starts listening again:

```
(defmodule messenger
 (export (print-result 0)))

(defun print-result ()
  (receive
    (msg
      (: io format '"Received message: '~s'~n" (list msg))
      (print-result))))
```

Let's take it for a test drive:

```
> (c '"messenger")
#(module messenger)
> (set pid (spawn 'messenger 'print-result ()))
<0.55.0>
> (! pid '"Zaphod was here.")
"Zaphod was here."
Received message: 'Zaphod was here.'
> (! pid '"Ford is missing.")
"Ford is missing."
Received message: 'Ford is missing.'
> (! pid '"Arthur is pining for Trillian.")
"Arthur is pining for Trillian."
Received message: 'Arthur is pining for Trillian.'
```

Horray! You've just written a simple listener in LFE!

3 Process Registry in Erlang/LFE

We've been setting the `pid` variable in the REPL so that we don't have to call `(self)` repeatedly. However, you don't *have* to use the pid. If you have a function running in a process whose purpose is to act as a service for any number of other processes, you may find the Erlang process registration system just the thing for you.

There are a handful of processes that are registered by the shell. We can see them by doing the following:

```
 > (: c regs)

** Registered procs on node nonode@nohost **
Name                  Pid          Initial Call                      Reds Msgs
application_controlle <0.6.0>      erlang:apply/2                     434    0
code_server           <0.18.0>     erlang:apply/2                  205428    0
erl_prim_loader       <0.3.0>      erlang:apply/2                  633205    0
error_logger          <0.5.0>      gen_event:init_it/6               4973    0
file_server_2         <0.17.0>     file_server:init/1                 660    0
global_group          <0.16.0>     global_group:init/1                 59    0
global_name_server    <0.12.0>     global:init/1                       50    0
inet_db               <0.15.0>     inet_db:init/1                     851    0
init                  <0.0.0>      otp_ring0:start/2                 4663    0
kernel_safe_sup       <0.24.0>     supervisor:kernel/1                 58    0
kernel_sup            <0.10.0>     supervisor:kernel/1              37499    0
rex                   <0.11.0>     rpc:init/1                          35    0
standard_error        <0.20.0>     erlang:apply/2                       9    0
standard_error_sup    <0.19.0>     supervisor_bridge:standar           41    0
user                  <0.22.0>     erlang:apply/2                    8009    0
user_drv              <0.28.0>     user_drv:server/2                52096    0

** Registered ports on node nonode@nohost **
Name                  Id              Command
ok
>
```

Let's spawn our previously-defined function and register it:

```
> (set pid-2 (spawn 'messenger 'print-result ()))
<0.52.0>
> (register 'msg-svc pid-2)
true
>
```

Now, if we call `(: c regs)` again, we'll see our newly-registered process in the table:

```
'msg-svc'             <0.52.0>     messenger:'print-result'/            1    0
```

We see that it's running; let's do a lookup on it by name to get the pid:

```
> (whereis 'msg-svc)
<0.52.0>
>
```

Just to confirm that this is the same process we spawned:

```
> pid-2
<0.52.0>
```

We've poked around a bit, now let's acutally use the new atom we've set as the reference for our messaging service:

```
> (! 'msg-svc '"This is Prostetnic Vogon Jeltz...")
"This is Prostetnic Vogon Jeltz..."
Received message: 'This is Prostetnic Vogon Jeltz...'
>
```

That about wraps it up for the process registry!

4 Process Communications

4.1 It's a Two Way Street

We've spent some time looking at sending messages to processes manually. Now let's see how processes can send messages to each other. This has the practical implication of being able to report back on something from one process to another process which might have initiated the current action.

For instance, if we could have a function that spawns several lightweight processes, each of which may return data and a status code. If these spawned processes have the calling process' ID, then they can, in turn, send their results back to the caller so that the data and statuses may be reported upon.

4.2 Walk the Talk

To demonstrate this, let's update our `print-result` function to do this:

- pattern match for a process id as well as a message
- write message about sending data to another process
- actually send that data to the other process

Additionally, let's create a new function that `spawn`s the `print-result` listener and then sends a message to it.

Save the following as `messenger-back.lfe`:

```
(defmodule messenger-back
 (export (print-result 0) (send-message 1)))

(defun print-result ()
  (receive
    ((tuple pid msg)
      (: io format '"Received message: '~s'~n" (list msg))
      (: io format '"Sending message to process ~p ...~n" (list pid))
      (! pid (tuple msg))
      (print-result))))

(defun send-message (calling-pid msg)
  (let ((spawned-pid (spawn 'messenger-back 'print-result ())))
    (! spawned-pid (tuple calling-pid msg))))
```

With these in place, let's start up our LFE REPL and try it out. First we'll compile our code and then kick things off with our `send-message` function. Note that this will `spawn` a listener and send it a message:

```
> (c '"messenger-back")
#(module messenger-back)
> (: messenger-back send-message '"There was a terrible ghastly silence.")
#(<0.25.0> "There was a terrible ghastly silence.")
Received message: 'There was a terrible ghastly silence.'
Sending message to process <0.25.0> ...
>
```

Let's take a look at shell's message queue (the quick and dirty way) to see if, in fact, the message *was* sent to the shell process:

```
> (: c flush)
Shell got {"There was a terrible ghastly silence."}
ok
>
```

Sure enough, it was :-)

4.3 Conclusion

Well, that about wraps it up. We may add more information about working with Erlang lightweight processes in LFE, but the material covered so far in this tutorial should give you a solid foundation for exploring the ways in which you can use processes to breathe life into your applications.