Executing TEX

Abstract
Much of the LUA code in CONTEXT originates from experiments. When it survives in the source code it is probably used, waiting to be used or kept for educational purposes. The functionality that we describe here has already been present for a while in CONTEXT, but improved a little starting with LUATEX 1.08 due to an extra helper. The code shown here is generic and not used in CONTEXT as such.

Say that we have this code:

```
for i=1,10000 do
    tex.sprint("1")
    tex.sprint("2")
    for i=1,3 do
        tex.sprint("3")
        tex.sprint("4")
        tex.sprint("5")
    end
    tex.sprint("\\space")
end
```

When we call `\directlua` with this snippet we get some 30 pages of `12345345345`. The printed text is saved till the end of the LUA call, so basically we pipe some 170.000 characters to TEX that get interpreted as one paragraph.

Now imagine this:

```
\setbox0\hbox{xxxxxxxxxxx} \number\wd0
```

which gives 3595950. If we check the box in LUA, with:

```
tex.sprint(tex.box[0].width)
tex.sprint("\\enspace")
tex.sprint("\\setbox0\\hbox{!}")
tex.sprint(tex.box[0].width)
```

the result is `3595950 3595950`, which is not what you would expect at first sight. However, if you consider that we just pipe to a TEX buffer that gets parsed after the LUA call, it will be clear that the reported width is the width that we started with. It will work all right if we say:

```
tex.sprint(tex.box[0].width)
tex.sprint("\\enspace")
tex.sprint("\\setbox0\\hbox{!}")
tex.sprint("\\directlua{tex.sprint(tex.box[0].width)}")
```

because now we get: `3595950 301500`. It's not that complex to write some support code that makes this more convenient. This can work out quite well but there is a drawback. If we use this code:

```
print(status.input_ptr)
tex.sprint(tex.box[0].width)
tex.sprint("\\enspace")
tex.sprint("\\setbox0\\hbox{!}")
tex.sprint("\\directlua{print(status.input_ptr)\
    tex.sprint(tex.box[0].width)}")
```

Here we get `6` and `7` reported. You can imagine that when a lot of nested `\directlua` calls happen, we can get an overflow of the input level or (depending on what we do) the input stack size. Ideally we want to do a LUA call, temporarily go to TEX, return to LUA, etc. without needing to worry about nesting and possible crashes due to LUA itself running into problems. One charming solution is to use so-called coroutines: independent LUA threads that one can switch between — you jump out from the current routine to another and from there back to the current one. However, when we use `\directlua` for that, we still have this nesting issue and what is worse, we keep nesting function calls too. This can be compared to:

```
\def\whatever{\ifdone\whatever\fi}
```

where at some point `\ifdone` is false so we quit. But we keep nesting when the condition is met, so eventually we can end up with some nesting related overflow. The following:

```
\def\whatever{\ifdone\expandafter\whatever\fi}
```

is less likely to overflow because there we have tail recursion which basically boils down to not nesting but continuing. Do we have something similar in LUATEX for LUA? Yes, we do. We can register a function, for instance:

```
lua.get_functions_table()[1] = function() print("Hi there!") end
```

and call that one with:

```
\luafunction 1
```

This is a bit faster than calling a function like:

```
\directlua{HiThere()}
```

which can also be achieved by

```
\directlua{print("Hi there!")}
```

which sometimes can be more convenient. Anyway, a function call is what we can use for our purpose as it doesn't involve interpretation and effectively behaves like a tail call. The following snippet shows what we have in mind:

```
tex.routine(function()
    tex.sprint(tex.box[0].width)
    tex.sprint("\\enspace")
    tex.sprint("\\setbox0\\hbox{!}")
    tex.yield()
    tex.sprint(tex.box[0].width)
end)
```

We start a routine, jump out to TEX in the middle, come back when we're done and continue. This gives us: 3595950 188640, which is what we expect.
3595950 188640
This mechanism permits efficient (nested) loops like:

```
tex.routine(function()
    for i=1,10000 do
        tex.sprint("1")
        tex.yield()
        tex.sprint("2")
        tex.routine(function()
            for i=1,3 do
```

```
                tex.sprint("3")
                tex.yield()
                tex.sprint("4")
                tex.yield()
                tex.sprint("5")
            end
        end)
        tex.sprint("\\space")
        tex.yield()
    end
end)
```

We do create coroutines, go back and forwards between LUA and TEX, but avoid memory being filled up with printed content. If we flush paragraphs (instead of e.g. the space) then the main difference is that instead of a small delay due to the loop unfolding in a large set of prints and accumulated content, we now get a steady flushing and processing.

However, we can still have an overflow of input buffers because we still nest them: the limitation at the TEX end has moved to a limitation at the LUA end. How come? Here is the code that we use:

```
local stepper = nil
local stack   = { }
local fid     = 0xFFFFFF
local goback  = "\\luafunction" .. fid .. "\\relax"

function tex.resume()
    if coroutine.status(stepper) == "dead" then
        stepper = table.remove(stack)
    end
    if stepper then
        coroutine.resume(stepper)
    end
end

lua.get_functions_table()[fid] = tex.resume

function tex.yield()
    tex.sprint(goback)
    coroutine.yield()
    texio.closeinput()
end

function tex.routine(f)
    table.insert(stack,stepper)
    stepper = coroutine.create(f)
    tex.sprint(goback)
end
```

The `routine` creates a coroutine, and `yield` gives control to TEX. The `resume` is done at the TEX end when we're finished there. In practice this works fine and when you permit enough nesting and levels in TEX then you will not easily overflow.

When I picked up this side project and wondered how to get around it, it suddenly struck me that if we could just quit the current input level then nesting would not be a problem. Adding a simple helper to the engine made that possible (of course figuring it out took a while):

```
local stepper = nil
local stack   = { }
local fid     = 0xFFFFFF
local goback  = "\\luafunction" .. fid .. "\\relax"

function tex.resume()
    if coroutine.status(stepper) == "dead" then
        stepper = table.remove(stack)
    end
    if stepper then
        coroutine.resume(stepper)
    end
end

lua.get_functions_table()[fid] = tex.resume

if texio.closeinput then
    function tex.yield()
        tex.sprint(goback)
        coroutine.yield()
        texio.closeinput()
    end
else
    function tex.yield()
        tex.sprint(goback)
        coroutine.yield()
    end
end

function tex.routine(f)
    table.insert(stack,stepper)
    stepper = coroutine.create(f)
    tex.sprint(goback)
end
```

The trick is in `texio.closeinput`, a recent helper and one that should be used with care. We assume that the user knows what she or he is doing. On an old laptop with a i7-3840 processor running WINDOWS 10 the following snippet takes less than 0.35 seconds with LUATEX and 0.26 seconds with LUAJITTEX.

```
tex.routine(function()
    for i=1,10000 do
        tex.sprint("\\setbox0\\hpack{x}")
        tex.yield()
        tex.sprint(tex.box[0].width)
        tex.routine(function()
            for i=1,3 do
                tex.sprint("\\setbox0\\hpack{xx}")
                tex.yield()
                tex.sprint(tex.box[0].width)
            end
        end)
    end
end)
```

Say that we run the bad snippet:

```
for i=1,10000 do
    tex.sprint("\\setbox0\\hpack{x}")
    tex.sprint(tex.box[0].width)
    for i=1,3 do
```

```
        tex.sprint("\\setbox0\\hpack{xx}")
        tex.sprint(tex.box[0].width)
    end
end
```

This time we need 0.12 seconds in both engines. So what if we run this:

```
\dorecurse{10000}{%
    \setbox0\hpack{x}
    \number\wd0
    \dorecurse{3}{%
        \setbox0\hpack{xx}
        \number\wd0
    }%
}
```

Pure TEX needs 0.30 seconds for both engines but there we lose 0.13 seconds on the loop code. In the LUA example where we yield, the loop code takes hardly any time. As we need only 0.05 seconds more it demonstrates that when we use the power of LUA the performance hit of the switch is quite small: we yield 40.000 times! In general, such differences are far exceeded by the overhead: the time needed to typeset the content (which `\hpack` doesn't do), breaking paragraphs into lines, constructing pages and other overhead involved in the run. In CONTEXT we use a slightly different variant which has 0.30 seconds more overhead, but that is probably true for all LUA usage in CONTEXT, but again, it disappears in other runtime.

Here is another example:

```
\def\TestWord#1%
  {\directlua{
     tex.routine(function()
       tex.sprint("\\setbox0\\hbox{\\tttf #1}")
       tex.yield()
       tex.sprint(math.round(100 * tex.box[0].width/tex.hsize))
       tex.sprint(" percent of the hsize: ")
       tex.sprint("\\box0")
     end)
  }}
```

```
The width of next word is \TestWord {inline}!
```

The width of next word is 8 percent of the hsize: `inline`!

Now, in order to stay realistic, this macro can also be defined as:

```
\def\TestWord#1%
  {\setbox0\hbox{\tttf #1}%
   \directlua{
      tex.sprint(math.round(100 * tex.box[0].width/tex.hsize))
   } %
   percent of the hsize: \box0\relax}
```

We get the same result: "The width of next word is 8 percent of the hsize: `inline`!".

We have been using a LUA-TEX mix for over a decade now in CONTEXT, and have never really needed this mixed model. There are a few places where we could (have) benefited from it and we might use it in a few places, but so far we have done fine without it. In fact, in most cases typesetting can be done fine at the TEX end. It's all a matter of imagination.

Hans Hagen