Executing \TeX in Lua: Coroutines

Hans Hagen

Much of the Lua code in Con\TeXt 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 Con\TeXt, but improved a little starting with Lua\TeX 1.08 due to an extra helper. The code shown here is generic and not used in Con\TeXt as such.

Say that we have this code:

\begin{verbatim}
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(\hspace{1em})
end
\end{verbatim}

When we call \texttt{\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:

\begin{verbatim}
\setbox0\hbox{xxxxxxxxxxx} \number\wd0
\end{verbatim}

which gives 4461336. If we check the box in Lua:

\begin{verbatim}
tex.sprint(tex.box[0].width)  
tex.sprint(\enspace)  
tex.sprint(\setbox0\hbox{!})  
tex.sprint(tex.box[0].width)
\end{verbatim}

the result is 4461336 4461336, 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:

\begin{verbatim}
tex.sprint(tex.box[0].width)  
tex.sprint(\enspace)  
tex.sprint(\setbox0\hbox{!})  
tex.sprint(\directlua{\tex.sprint(tex.box[0].width)})
\end{verbatim}

Now imagine this:

\begin{verbatim}
\def\whatever{\ifdone\whatever\fi}
\end{verbatim}

where at some point \texttt{\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:

\begin{verbatim}
\def\whatever{\ifdone\expandafter\whatever\fi}
\end{verbatim}

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 Lua\TeX for Lua? Yes, we do. We can register a function, for instance

\begin{verbatim}
lua.get_functions_table()[1] = function() print("Hi there!") end
\end{verbatim}

and call that one with:

\begin{verbatim}
\luafunction 1
\end{verbatim}

This is a bit faster than calling a function like:

\begin{verbatim}
\directlua{HiThere()}
\end{verbatim}

which can also be achieved by

\begin{verbatim}
\directlua{print("Hi there!")}
\end{verbatim}

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:

\begin{verbatim}
tex.routine(function()  
tex.sprint(tex.box[0].width)  
tex.sprint(\enspace)  
tex.sprint(\setbox0\hbox{!})  
tex.sprint(\directlua{\tex.sprint(\hspace{1em})})
end)
\end{verbatim}

because now we get: 4461336 443625. 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:

\begin{verbatim}
print(status.input_ptr)  
tex.sprint(tex.box[0].width)  
tex.sprint(\enspace)  
tex.sprint(\setbox0\hbox{!})  
tex.sprint(\directlua{\print(status.input_ptr)\hspace{1em}})
\end{verbatim}

Here we get 6 and 7 reported. You can imagine that when a lot of nested \texttt{\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 \texttt{\directlua} for that, we still have this nesting issue and what is worse, we keep nesting function calls too. This can be compared to:

\begin{verbatim}
\def\whatever{\ifdone\whatever\fi}
\end{verbatim}

which at some point \texttt{\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:

\begin{verbatim}
\def\whatever{\ifdone\expandafter\whatever\fi}
\end{verbatim}

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 Lua\TeX for Lua? Yes, we do. We can register a function, for instance

\begin{verbatim}
lua.get_functions_table()[1] = function() print("Hi there!") end
\end{verbatim}

and call that one with:

\begin{verbatim}
\luafunction 1
\end{verbatim}

This is a bit faster than calling a function like:

\begin{verbatim}
\directlua{HiThere()}
\end{verbatim}

which can also be achieved by

\begin{verbatim}
\directlua{print("Hi there!")}
\end{verbatim}

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:

\begin{verbatim}
tex.routine(function()  
tex.sprint(tex.box[0].width)  
tex.sprint(\enspace)  
tex.sprint(\setbox0\hbox{!})  
tex.sprint(\directlua{\tex.sprint(\hspace{1em})})
end)
\end{verbatim}

because now we get: 4461336 443625. 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:

\begin{verbatim}
print(status.input_ptr)  
tex.sprint(tex.box[0].width)  
tex.sprint(\enspace)  
tex.sprint(\setbox0\hbox{!})  
tex.sprint(\directlua{\print(status.input_ptr)\hspace{1em}})
\end{verbatim}

We start a routine, jump out to \TeX in the middle, come back when we’re done and continue. This gives us: 4461336 218508, which is what we expect.

This mechanism permits efficient (nested) loops like: 41

Executing \TeX in Lua: Coroutines
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:

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

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):

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

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

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 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 i7-3840QM processor running Windows 10 the following snippet takes less than 0.35 seconds with Lua\TeX and 0.26 seconds with LuaJIT\TeX.

```markdown
tex.routine(function()
  for i=1,10000 do
    tex.sprint("\setbox0\hpack{x}")
    tex.yield()
  end
end

Hans Hagen
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 9 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 9 percent of the hsize: inline!”.

We have been using a Lua–\TeX{} mix for over a decade now in Con\TeX{}t, and have never really needed this mixed model. There are a few places where we could (have) benefitted 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.

\include{execute_tex_lua_coroutines}

\HansHagen
Pragma ADE
http://pragma-ade.com

Executing \TeX{} in Lua: Coroutines