workflow

support in context

Contents

Introduction

This manual contains some information about features that can help you to manage workflows or CONTEXT related processes. Because we use CONTEXT ourselves all that we need ends up in the distribution. When you discover something workflow related that is not yet covered here, you can tell me. I simply forget about all there is, especially if it's made for projects. Don't expect this manual to be complete or extensive, it's just a goodie.

Hans Hagen,
PRAGMA ADE
Hasselt NL
August 2021

1 Running

1.1 Errors

A CONTEXT runs normally spits out quite some information to the console. In fact, even more information can go to the log file. It makes sense to have a look at the end of the log file occasionally because there you can find information about the (file) structure loaded, modules, issues with references and/or fonts, etc.

One problem with the terminal is that you can miss an issue easily, but there is a way out of this:

```
\enabledirectives[logs.errors]
```

The command line argument `--errors` has the same consequence. If you want to quit in an error, you can say for instance:

```
\enabledirectives[logs.errors=missing characters]
```

If you run CONTEXT with `--trackers` or `--directives` you get some information about the possible extra tracing. It might be illustrative to run a file with:

```
\enabletrackers[*]
```

There are all kind of trackers and directives and you can get a list with:

```
context --trackers --directives
```

An example is:

```
\enabletrackers[fonts.missing=replace]
```

or just:

```
\enabletrackers[fonts.missing]
```

1.2 Silent

The `--silent` options blocks most message. You can also pass a list (or pattern) of categories to silence. The `--noconsole` option only disables logging to the console. The error reporting mentioned in the previous section will never be silenced.

2 Accessing resources

One of the benefits of TEX is that you can use it in automated workflows where large quantities of data is involved. A document can consist of several files and normally also includes images. Of course there are styles involved too. At PRAGMA ADE normally put styles and fonts in:

```
/data/site/context/tex/texmf-project/tex/context/user/<project>/...
/data/site/context/tex/texmf-fonts/data/<foundry>/<collection>/...
```

alongside

```
/data/framework/...
```

where the job management services are put, while we put resources in:

```
/data/resources/...
```

The processing happens in:

```
/data/work/<uuid user space>/
```

Putting styles (and resources like logos and common images) and fonts (if the project has specific ones not present in the distribution) in the TEX tree makes sense because that is where such files are normally searched. Of course you need to keep the distributions file database upto-date after adding files there.

Processing has to happen isolated from other runs so there we use unique locations. The services responsible for running also deal with regular cleanup of these temporary files.

Resources are somewhat special. They can be stable, i.e. change seldom, but more often they are updated or extended periodically (or even daily). We're not talking of a few files here but of thousands. In one project we have 20 thousand resources, that can be combined into arbitrary books, and in another one, each chapter alone is about 400 XML and image files. That means we can have 5000 files per book and as we have at least 20 books, we end up with 100K files. In the first case accessing the resources is easy because there is a well defined structure (under our control) so we know exactly where each file sits in the resource tree. In the 100K case there is a deeper structure which is in itself predictable but because many authors are involved the references to these files are somewhat instable (and undefined). It is surprising to notice that publishers don't care about filenames (read: cannot control all the parties involved) which means that we have inconsistent use of mixed case in filenames, and spaces, underscores and dashes creeping in. Because typesetting for paper is always at the end of the pipeline (which nowadays is mostly driven by (limitations) of web products) we need to have a robust and flexible lookup mechanism. It's a side effect of the click and point culture: if objects are associated (filename in source file with file on the system) anything you key in will work, and consistency completely depends

on the user. And bad things then happen when files are copied, renamed, etc. In that stadium we can better be tolerant than try to get it fixed.[1]

```
foo.jpg
bar/foo.jpg
images/bar/foo.jpg
images/foo.jpg
```

The xml files have names like:

```
b-c.xml
a/b-c.jpg
a/b/b-c.jpg
a/b/c/b-c.jpg
```

So it's sort of a mess, especially if you add arbitrary casing to this. Of course one can argue that a wrong (relative) location is asking for problems, it's less an issue here because each image has a unique name. We could flatten the resource tree but having tens of thousands of files on one directory is asking for problems when you want to manage them.

The typesetting (and related services) run on virtual machines. The three directories:

```
/data/site
/data/resources
/data/work
```

are all mounted as nfs shares on a network storage. For the styles (and binaries) this is no big deal as normally these files are cached, but the resources are another story. Scanning the complete (mounted) resource tree each run is no option so there we use a special mechanism in CONTEXT for locating files.

Already early in the development of MKIV one of the locating mechanisms was the following:

```
tree:////data/resources/foo/**/drawing.jpg
tree:////data/resources/foo/**/Drawing.jpg
```

Here the tree is scanned once per run, which is normally quite okay when there are not that many files and when the files reside on the machine itself. For a more high

[1] From what we normally receive we often conclude that copy-editing and image production companies don't impose any discipline or probably simply lack the tools and methods to control this. Some of our workflows had checkers and fixers, so that when we got 5000 new resources while only a few needed to be replaced we could filter the right ones. It was not uncommon to find duplicates for thousands of pictures: similar or older variants.

performance approach using network shares we have a different mechanism. This time it looks like this:

```
dirlist:/data/resources/**/drawing.jpg
dirlist:/data/resources/**/Drawing.jpg
dirlist:/data/resources/**/just/some/place/drawing.jpg
dirlist:/data/resources/**/images/drawing.jpg
dirlist:/data/resources/**/images/drawing.jpg?option=fileonly
dirfile:/data/resources/**/images/drawing.jpg
```

The first two lookups are wildcard. If there is a file with that name, it will be found. If there are more, the first hit is used. The second and third examples are more selective. Here the part after the `**` has to match too. So here we can deal with multiple files named `drawing.jpg`. The last two equivalent examples are more tolerant. If no explicit match is found, a lookup happens without being selective. The case of a name is ignored but when found, a name with the right case is used.

You can hook a path into the resolver for source files, for example:

```
\usepath                          [dirfile://./resources/**]
\setupexternalfigures[directory=dirfile://./resources/**]
```

You need to make sure that file(name)s in that location don't override ones in the regular TEX tree. These extra paths are only used for source file lookups so for instance font lookups are not affected.

When you add, remove or move files the tree, you need to remove the `dirlist.*` files in the root because these are used for locating files. A new file will be generated automatically. Don't forget this!

When content doesn't change an alternative discussed in in a later chapter can be considered: hashed databases of files.

3 Graphics

3.1 Bad names

After many years of using CONTEXT in workflows where large amounts of source files as well as graphics were involved we can safely say that it's hard for publishers to control the way these are named. This is probably due to the fact that in a click-and-point based desktop publishing workflow names don't matter as one stays on one machine, and names are only entered once (after that these names become abstractions and get cut and pasted). Proper consistent resource managament is simply not part of the flow.

This means that you get names like:

```
foo_Bar_01_03-a.EPS
foo__Bar-01a_03.eps
foo__Bar-01a_03.eps
foo BarA  01-03.eps
```

Especially when a non proportional screen font is used multiple spaces can look like one. In fancy screen fonts upper and lowercase usage might get obscured. It really makes one wonder if copy-editing or adding labels to graphics isn't suffering from the same problem.

Anyhow, as in an automated rendering workflow the rendering is often the last step you can imagine that when names get messed up it's that last step that gets blamed. It's not that hard to sanitize names of files on disk as well as in the files that refer to them, and we normally do that we have complete control. This is no option when all the resources are synchronzied from elsewhere. In that case the only way out is signaling potential issues. Say that in the source file there is a reference:

```
foo_Bar_01_03-a.EPS
```

and that the graphic on disk has the same name, but for some reason after an update has become:

```
foo-Bar_01_03-a.EPS
```

The old image is probably still there so the update is not reflected in the final product. This is not that uncommon when you deal with tens of thousands of files, many editors and graphic designers, and no strict filename policy.

For this we provide the following tracing option:

```
\enabletrackers[graphics.lognames]
```

This will put information in the log file about included graphics, like:

```
system          > graphics > start names
```

```
used graphic    > asked   : cow.pdf
used graphic    > comment : not found
used graphic    > asked   : t:/sources/cow.pdf
used graphic    > format  : pdf
used graphic    > found   : t:/sources/cow.pdf
used graphic    > used    : t:/sources/cow.pdf

system          > graphics > stop names
```

You can also add information to the file itself:

```
\usemodule[s-figures-names]
```

Of course that has to be done at the end of the document. Bad names are reported and suitable action can be taken.

3.2 Downsampling

You can plug in you rown converter, here is an example:

```
\startluacode

figures.converters.jpg = figures.converters.jpg or { }

figures.converters.jpg["lowresjpg.pdf"] =
  function(oldname,newname,resolution)
    figures.programs.run (
      [[gm]],
      [[convert -geometry %nx%x%ny% -compress JPEG "%old%" "%new%"]],
      {
         old = old,
         new = new,
         nx  = resolution or 300,
         ny  = resolution or 300,
      }
    )
  end
\stopluacode
```

You can limit the search to a few types and set the resolution with:

```
\setupexternalfigures
  [order={pdf,jpg},
   resolution=100,
```

```
   method=auto]
```

And use it like:

```
\externalfigure[verybig.jpg][height=10cm]
```

The second string passed to the `run` helper contains the arguments to the first one. The variables between percent signs get replaced by the variables in the tables passed as third argument.

3.3 Trackers

If you want a lot of info you can say:

```
\enabletrackers[figures.*]
```

But you can be more specific. With `graphics.locating` you will get some insight in where files are looked for. The `graphics.inclusion` tracker gives some more info about actual inclusion. The `graphics.bases` is kind of special and only makes sense when you use the graphic database options. The `graphics.conversion` and related tracker `graphics.programs` show if and how conversion of images takes place.

The `graphics.lognames` will make sure that some extra information about used graphics is saved in the log file, The `graphics.usage` tracker will produce a file `<jobname>-figures-usage.lua` that contains information about found (or not found) images and the way they are used.

3.4 Compression

A PNG image uses several methods for compression. The image data itself can be compacted by taking pixels around each individual pixel into account. Storing deltas instead of absolute values can for instance result in strips of zeros. These in turn compress well using zlib compression. Each scanline starts with a filterbyte that indicates how to look at the surrounding pixels. While in PDF a JPEG image is included as-is, a PNG often take a bit more work. A (optional) mask has to be split off, as does an (optional) index. In the worst case we need to deinterlace. The PNG inclusion mechanism in CONTEXT LMTX takes care of this in an as efficient as possible way. Nevertheless there are some knobs you can turn:

```
\enabledirectives[graphics.png.recompress]
\enabledirectives[graphics.png.compresslevel=9]
```

A higher compress level makes the run somewhat smaller but also fives smaller files. The default compress level is 3. When an image doesn't need to be transformed (due

to mask, index or interlace), you have to force recompression with the `recompress` directive. The sample image `mill.png` has a size of `154,869` bytes.

```
\startTEXpage
    \externalfigure[mill.png]
\stopTEXpage
```

The next table shows the consequences of setting the directives. The runtime is of course dependent of the machine you run the sample on. If you have lots of images it might make sense to have a final run with a higher compress level. The PDF file has some extra overhead (like metadata and page related information).

	compression	**filesize**	**runtime**
default		156,964	0.516
recompress	3	144,418	0.531
compresslevel	0	281,071	0.516
compresslevel	9	137,375	0.547

4 Suspects

When many authors and editors are involved there is the danger of inconsistent spacing being applied. We're not only talking of the (often invisible in editing programs) nobreak spaces, but also spacing inside or outside quotations and before and after punctuation or around math.

In CONTEXT we have a built–in suspects checker that you can enable with the following command:

```
\enabletrackers[typesetters.suspects]
```

The next table shows some identified suspects.

`foo$x$`	fo$x$
`$x$bar`	$x$bar
`foo$x$bar`	fo$x$bar
`$f+o+o$:`	$f+o+o$:
`;$f+o+o$`	;$f+o+o$
`; bar`	; bar
`foo:bar`	foo:bar
`\quote{ foo }`	‘ foo’
`\quote{bar }`	‘bar’
`\quote{ bar}`	‘ bar’
`(foo )`	(foo)
`\{foo \}`	{foo }
`foo{\bf gnu}bar`	fo**gn**ubar
`foo{\it gnu}bar`	fo*gn*ubar
`foo$x^2$bar`	fox^2bar
`foo\nobreakspace bar`	foo bar

Of course this analysis is not perfect but we use it in final checking of complex documents that are generated automatically from XML.[2]

[2] Think of math school books where each book is assembled from over a thousands files.

5 Injectors

When you have no control over the source but need to manually tweak some aspects of the typesetting, like an occasional page break of column switch, you can use the injector mechanism. This mechanism is part of list and register building but can also be used elsewhere.

We have two buffers:

```
\startmixedcolumns[balance=yes]
  \dotestinjector{test}line 1 \par
  \dotestinjector{test}line 2 \par
  \dotestinjector{test}line 3 \par
  \dotestinjector{test}line 4 \par
  \dotestinjector{test}line 5
\stopmixedcolumns
```

and

```
\startmixedcolumns[balance=yes]
  \dotestinjector{test}line 1 \par
  \dotestinjector{test}line 2 \par
  \dotestinjector{test}line 3 \par
  \dotestinjector{test}line 4 \par
  \dotestinjector{test}line 5
\stopmixedcolumns
```

When typeset these come out as:

line 1
line 2
line 3

line 4
line 5

and

line 1
line 2
line 3

line 4
line 5

We can enable (and show) the injectors with:

```
\doactivateinjector{test} \showinjector
```

Now we get:

<1> **line 1**
<2> **line 2**
<3> **line 3**

<4> **line 4**
<5> **line 5**

and

<6> **line 1**
<7> **line 2**
<8> **line 3**
<9> **line 4**
<10> **line 5**

The small numbers are injector points. These will of course change when we add more in-between. Let's add actions to some of the injection points:

```
\setinjector[test][13][{\column}]
\setinjector[test][17][{\column}]
```

As expected we now get column breaks:

<11> **line 1**
<12> **line 2**
<13> **line 3**
<14> **line 4**
<15> **line 5**

and

<16> **line 1**
<17> **line 2**
<18> **line 3**
<19> **line 4**
<20> **line 5**

The next example is one you can run for your own. The `\showinjector` command is of course only enabled when the right injection points are to be identified. Normally setting injectors happens as last resort. We use it in automated XML workflows where last minute control is needed.

```
\showinjector

\setinjector[register][3][\column]
\setinjector[list]    [2][{\blank[3*big]}]

\starttext
    \placelist[section][criterium=text]
    \blank[3*big]
    \placeregister[index][criterium=text]
    \page
    \startsection[title=Alpha] first  \index{first}  \stopsection
    \startsection[title=Beta]  second \index{second} \stopsection
    \startsection[title=Gamma] third  \index{third}  \stopsection
    \startsection[title=Delta] fourth \index{fourth} \stopsection
\stoptext
```

6 XML

When you have an XML project with many files involved, finding the right spot of something that went wrong can be a pain. In one of our project the production of some 50 books involves 60.000 XML files and 20.000 images.[3] Say that we have the following file:

```
<?xml version='1.0'?>
<document>
  <include name="temp-01.xml"/> <include name="temp-02.xml"/>
  <include name="temp-03.xml"/> <include name="temp-04.xml"/>
  <include name="temp-05.xml"/> <include name="temp-06.xml"/>
  <include name="temp-07.xml"/> <include name="temp-08.xml"/>
  <include name="temp-09.xml"/> <include name="temp-10.xml"/>
</document>
```

Before we process this file we will merge the content of the files defined as includes into it. When this happens the filename is automatically registered so it can be accessed later.

```
\startxmlsetups xml:initialize
    \xmlincludeoptions{#1}{include}{filename|name}{recurse,basename}
    \xmlsetsetup{#1}{p|document}{xml:*}
\stopxmlsetups

\startxmlsetups xml:document
    \xmlflush{#1}
\stopxmlsetups

\startxmlsetups xml:p
    \inleftmargin{\infofont\xmlinclusion{#1}}
    \xmlflush{#1}
    \par
\stopxmlsetups

\xmlregistersetup{xml:initialize}

\xmlprocessbuffer{main}{demo}{}
```

In this case we put the name of the file in the margin. Depending on when and how elements are flushed other solutions, like overlays, can be used.

temp-01.xml snippet 1

temp-02.xml snippet 2

temp-03.xml snippet 3

[3] In the meantime we could trim this down a lot.

temp-04.xml snippet 4
temp-05.xml snippet 5
temp-06.xml snippet 6
temp-07.xml snippet 7
temp-08.xml snippet 8
temp-09.xml snippet 9
temp-10.xml snippet 10

At any moment you can see what the current node contains. The whole (merged) document is also available:

```
\xmlshow{main}
```

A small font is used to typeset the (sub)tree:

```
<?xml version='1.0'?>
<document>
  <p>snippet 1</p> <p>snippet 2</p>
  <p>snippet 3</p> <p>snippet 4</p>
  <p>snippet 5</p> <p>snippet 6</p>
  <p>snippet 7</p> <p>snippet 8</p>
  <p>snippet 9</p> <p>snippet 10</p>
</document>
```

You can also save the tree:

```
\xmlsave{main}{temp.xml}
```

This file looks like:

```
<?xml version='1.0'?>
<document>
  <p>snippet 1</p> <p>snippet 2</p>
  <p>snippet 3</p> <p>snippet 4</p>
  <p>snippet 5</p> <p>snippet 6</p>
  <p>snippet 7</p> <p>snippet 8</p>
  <p>snippet 9</p> <p>snippet 10</p>
</document>
```

7 Setups

Setups are a powerful way to organize styles. They are basically macros but live in their own namespace. One advantage is that spaces in a setup are ignored so you can code without bothering about spurious spaces. Here is a trick that you can use when one style contains directives for multiple products:

```
\startsetups tex:whatever
    \fastsetup{tex:whatever:\documentvariable{stylevariant}}
\stopsetups

\startsetups tex:whatever:foo
    FOO
\stopsetups

\startsetups tex:whatever:bar
    BAR
\stopsetups
```

Here we define a main setup `tex:whatever` that gets expanded in one of two variants, controlled by a document variable.

```
\setups{tex:whatever}

\setupdocument
  [stylevariant=foo]

\setups{tex:whatever}

\setupdocument
  [stylevariant=bar]

\setups{tex:whatever}
```

These lines result in:

FOO

BAR

In a similar fashion you can define XML setups that are used to render elements:

```
\startxmlsetups xml:whatever
    \xmlsetup{#1}{xml:whatever:\documentvariable{stylevariant}}
\stopxmlsetups

\startxmlsetups xml:whatever:foo
    FOO: \xmlflush{#1}
```

```
\stopxmlsetups

\startxmlsetups xml:whatever:bar
    BAR: \xmlflush{#1}
\stopxmlsetups
```

8 SYNCTEX

8.1 Introduction

Some users like the SYNCTEX feature that is built in the TEX engines. Personally I never use it because it doesn't work well with the kind of documents I maintain. If you have one document source, and don't shuffle around (reuse) text too much it probably works out okay but that is not our practice. Here I will describe how you can enable a more CONTEXT specific SYNCTEX support so that aware PDF viewers can bring you back to the source.

8.2 What we want

The SYNCTEX method roughly works as follows. Internally TEX constricts linked lists of glyphs, kerns, glue, boxes, rules etc. These elements are called nodes. Some nodes carry information about the file and line where they were created. In the backend this information gets somehow translated in a (sort of) verbose tree that describes the makeup in terms of boxes, glue and kerns. From that information the SYNCTEX parser library, hooked into a PDF viewer, can go back from a position on the screen to a line in a file. One would expect this to be a relative simple rectangle based model, but as far as I can see it's way more complex than that. There are some comments that CONTEXT is not supported well because it has a layered page model, which indicates that there are some assumptions about how macro packages are supposed to work. Also the used heuristics not only involve some specific spot (location) but also involve the corners and edges. It is therefore not so much a (simple) generic system but a mechanism geared for a macro package like LATEX.

Because we have a couple of users who need to edit complex sets of documents, coded in TEX or XML, I decided to come up with a variant that doesn't use the SYNCTEX machinery but manipulates the few SYNCTEX fields directly[4] and eventually outputs a straightforward file for the editor. Of course we need to follow some rules so that the editor can deal with it. It took a bit of trial and error to get the right information in the support file needed by the viewer but we got there.

The prerequisites of a decent CONTEXT “click on preview and goto editor” are the following:

[4] This is something that in my opinion should have been possible right from the start but it's too late now to change the system and it would not be used beyond CONTEXT anyway.

- It only makes sense to click on text in the text flow. Headers and footers are often generated from structure, and special typographic elements can originate in macros hooked into commands instead of in the source.
- Users should not be able to reach environments (styles) and other files loaded from the (normally read-only) TEX tree, like modules. We don't want accidental changes in such files.
- We not only have TEX files but also XML files and these can normally flush in rather arbitrary ways. Although the concept of lines is sort of lost in such a file, there is still a relation between lines and the snippets that make out the content of an XML node.
- In the case of XML files the overhead related to preserving line numbers should be minimal and have no impact on loading and memory when these features are not used.
- The overhead in terms of an auxiliary file size and complexity as well as producing that file should be minimal. It should be easy to turn on and off these features. (I'd never turn them on by default.)

It is unavoidable that we get more run time but I assume that for the average user that is no big deal. It pays off when you have a workflow when a book (or even a chapter in a book) is generated from hundreds of small XML files. There is no overhead when SYNCTEX is not used.

In CONTEXT we don't use the built-in SYNCTEX features, that is: we let filename and line numbers be set but often these are overloaded explicitly. The output file is not compressed and constructed by CONTEXT. There is no benefit in compression and the files are probably smaller than default SYNCTEX anyway.

8.3 Commands

Although you can enable this mechanism with directives it makes sense to do it using the following command.

```
\setupsynctex[state=start]
```

The advantage of using an explicit command instead of some command line option is that in an editor it's easier to disable this trickery. Commenting that line will speed up processing when needed. This command can also be given in an environment (style). On the command line you can say

```
context --synctex somefile.tex
```

A third method is to put this at the top of your file:

```
% synctex=yes
```

Often an XML files is very structured and although probably the main body of text is flushed as a stream, specific elements can be flushed out of order. In educational documents flushing for instance answers to exercises can happen out of order. In that case we still need to make sure that we go to the right spot in the file. It will never be 100% perfect but it's better than nothing. The above command will also enable XML support.

If you don't want a file to be accessed, you can block it:

```
\blocksynctexfile[foo.tex]
```

Of course you need to configure the viewer to respond to the request for editing. In Sumatra combined with SCITE the magic command is:

```
c:\data\system\scite\wscite\scite.exe "%f" "-goto:%l"
```

Such a command is independent of the macro package so you can just consult the manual or help info that comes with a viewer, given that it supports this linking back to the source at all.

8.4 Methods

Contrary to the native SYNCTEX we only deal with text which gives reasonable efficient output. If you enable tracing (see next section) you can what has become clickable. Instead of words you can also work with ranges, which not only gives less runtime but also much smaller `.synctex` files. Just try:

```
\setupsynctex[state=start,method=min]
```

to get words clickable and

```
\setupsynctex[state=start,method=max]
```

to get the more efficient ranges. The overhead for `min` is some 10 percent while `max` slows down around 5 percent.

8.5 Tracing

In case you want to see what gets synced you can enable a tracker:

```
\enabletrackers[system.synctex.visualize]
```

```
\enabletrackers[system.synctex.visualize=real]
```

The following tracker outputs some status information about XML flushing. Such trackers only make sense for developers.

```
\enabletrackers[system.synctex.xml]
```

8.6 Warning

Don't turn on this feature when you don't need it. This is one of those mechanism that hits performance badly.

Depending on needs the functionality can be improved and/or extended. Of course you can always use the traditional SYNCTEX method but don't expect it to behave as described here.

8.7 Two-way

In for instance the TEXshop editor, there is a two way connection. The nice thing about this editor is, is that it is also the first one to use the `mtx-synctex` script to resolve these links, instead of relying on a library. You can also use this script to inspect a SYNCTEX file yourself, The help into shows the possible directives.

```
mtxrun --script synctex
```

You can resolve positions in the PDF as well as in the sources and list all the known areas in the log.

9 Parallel processing

This is just a small intermezzo. Mid April 2020 Mojca asked on the mailing list how to best compile 5000 files, based on a template. The answer depends on the workflow and circumstances but one can easily come up with some factors that play a role.

- How complex is the document? How many pages are generated, how many fonts get used? Do we need multiple runs per document? Are images involved and if so, what format are they in? When processing relative small files we normally need seconds, not minutes.

- What machine is used? How powerful is the CPU, how many cores are available and how much memory do we have? Is the filesystem on a local SSD or on a remote file system? How well does file caching work? Again, we're talking seconds here.

- What engine is used? Assuming that MKIV is used, we can choose for LUATEX or LUAMETATEX. The former has faster backend code, the later a faster frontend. What is more efficient depends on the document. The later has some advantages that we will not mention here.

The tests mentioned below are run with a simple LUA script that manages the parallel runs. More about that later. As sample document we use this:

```
\setupbodyfont[dejavu]

\starttext
    \dorecurse{\getdocumentargument{noffiles}}{\input tufte\par}
\stoptext
```

We start with 100 runs of 10 inclusions. We permit 8 runs in parallel. A LUATEX run of 100 takes 32 seconds, a LUAJITTEX run uses 26 seconds, and LUAMETATEX does it in 25 seconds.[5] An interesting observation is memory consumption: LUAJITTEX, which has a different virtual machine and a limited memory model, peaks at 0.8GB for the eight parallel runs. The LUAMETATEX engine has the same demands. However, LUATEX needs 1.2GB. Bumping to 20 inclusions increased the runtime a few seconds for each engine.

The differences can be explained by a faster startup time of LUAMETATEX; for instance we don't use a compressed format (dump), but there are some other optimizations too, and even when they're close to unmeasurable, they might add up. The LUAJITTEX engine speeds up LUA interpretation which is reflected in runtime because CONTEXT spends half its time in LUA.

[5] I used a mingw cross compiled 64 bit binary; the GCC9 version seems somewhat slower than the previous compiler version.

As a next test I decided to run the test file 5000 times: Mojca's scenario. Including 10 sample files (per run) for those 5000 files took 1320 seconds. When we cache the included file we gain some 5 percent.

Does it matter how many jobs we run in parallel? The 2013 laptop I used for testing has four real cores that hyperthread to eight cores.[6] On 1000 jobs we need 320 seconds for 1000 files (10 inclusions) when we use four cores. With six cores we need 270 seconds, which is much better. With eight cores we go down to 260 seconds and ten cores, which is two more than there are, we get about the same runtime.[7] A TEX program is a single core process and it makes no sense to use more cores than the CPU provides.

```
\setupbodyfont[dejavu]

\starttext
    \dorecurse{\getdocumentargument{noffiles}}{\samplefile{tufte}\par}
\stoptext
```

Again, caching the input file as above saves a little bit: 10 seconds, so we get 250 seconds. When you run these tests on the machine that you normally work on, waiting for that many jobs to finish is no fun, so what if we (as I then normally do) watch some music video? With a full screen high resolution video shown in the foreground the runtime didn't change: still 250 seconds for 1000 jobs with eight parallel runs. On the other hand, a test with Firefox, which is quite demanding, running a video in the background, made the runtime going up by 30 seconds to 280. So, when doing some networking, decompression, all kinds of unknown tracking using JAVASCRIPT, etc. and therefore its own demands on cores and memory you might want to limit the number of parallel runs. These tests are probably not that meaningful but a good distraction when in lock down.

I'm still not sure if I should come up with a script for managing these parallel runs. But one thing I have added to the `context` runner is the (for now undocumented) option

```
--wipebusy
```

which, after a run removes the file

```
context-is-busy.tmp
```

This permits a management script to check if a run is done. Before starting a run (in a separate process) the script can write that file and by just checking if it is still there, the management script can decide when a next run can be started.

[6] The machine has an Intel i7-3840QM CPU, 16GB of memory and a 512 GB Samsung Pro SSD.
[7] On a more modern system, let alone a desktop computer, I expect these numbers to be much lower.

10 Hashed files

In a (basically free content) project we had to deal with tens of thousands of files. Most are in XML format, but there are also thousands of PNG, JPG and SVG images. In a large project like this, which covers a large part of Dutch school math, images can be shared. All the content is available for schools as HTML but can also be turned into printable form and because schools want to have stable content over specified periods one has to make a regular snapshot of this corpus. Also, distributing a few gigabytes of data is not much fun.

So, in order to bring the amount down a dedicated mechanism for handling files has been introduced. After playing with a SQLITE database we finally settled on just LUA, simply because it was faster and it also makes the solution independent.

The process comes down to creating a file database once in a while, loading a relatively small hash mapping at runtime and accessing files from a large data file on demand. Optionally files can be compressed, which makes sense for the textual files.

A database is created with one of the CONTEXT extras, for instance:

```
context --extra=hashed --database=m4 --pattern=m4all/**.xml --compress
context --extra=hashed --database=m4 --pattern=m4all/**.svg --compress
context --extra=hashed --database=m4 --pattern=m4all/**.jpg
context --extra=hashed --database=m4 --pattern=m4all/**.png
```

The database uses two files: a small `m4.lua` file (some 11 megabytes) and a large `m4.dat` (about 820 megabytes, coming from 1850 megabytes originals). Alternatively you can use a specification, say `m4all.lua`:

```
return {
    { pattern  = "m4all/**.xml$", compress = true  },
    { pattern  = "m4all/**.svg$", compress = true  },
    { pattern  = "m4all/**.jpg$", compress = false },
    { pattern  = "m4all/**.png$", compress = false },
}

context --extra=hashed --database=m4 --patterns=m4all.lua
```

You should see something like this on the console:

```
hashed > database 'hasheddata', 1627 paths, 46141 names,
    36935 unique blobs, 29674 compressed blobs
```

So here we share some ten thousand files (all images). In case you wonder why we keep the duplicates: they have unique names (copies) so that when a section is updated there is no interference with other sections. The tree structure is mostly six deep (sometimes there is an additional level).

Accessing files is the same as with files on the system, but one has to register a database first:

```
\registerhashedfiles[m4]
```

A fully qualified specifier looks like this (not too different from other specifiers):

```
\externalfigure
  [hashed:///m4all/books/chapters/h3/h3-if1/images/casino.jpg]
\externalfigure
  [hashed:///m4all/books/chapters/ha/ha-c4/images/ha-c44-ex2-s1.png]
```

but nicer would be :

```
\externalfigure
  [m4all/books/chapters/h3/h3-if1/images/casino.jpg]
\externalfigure
  [m4all/books/chapters/ha/ha-c4/images/ha-c44-ex2-s1.png]
```

This is possible when we also specify:

```
\registerfilescheme[hashed]
```

This makes the given scheme based resolver kick in first, while the normal file lookup is used as last resort.

This mechanism is written on top of the infrastructure that has been part of CONTEXT MKIV right from the start but this particular feature is only available in LMTX (backporting is likely a waste of time).

Just for the record: this mechanism is kept simple, so the database has no update and replace features. One can just generate a new one. You can test for a valid database and act upon the outcome:

```
\doifelsevalidhashedfiles {m4} {
    \writestatus{hashed}{using hashed data}
    \registerhashedfiles[m4]
    \registerfilescheme[hashed]
} {
    \writestatus{hashed}{no hashed data}
}
```

Future versions might introduce filename normalization (lowercase, cleanup) so consider this as a first step. First we need test it for a while.