A Spreadsheet and a Debugger walk into a Shell

Posted on September 16, 2024 by bjornstahl

Here we continue the series of posts on the development of a command-line shell which defies terminal emulation by using the display server API locally and a purpose built network protocol remotely.

Previous episodes include: The Day of a new Command-Line Interface: Shell (high level) Whipping up a new Shell – Lash#Cat9 (technical demo), Cat9 Microdosing: Stash and List (feature highlight), Cat9 Microdosing: Each and Contain (feature highlight)

For this round we have added an interactive spreadsheet representation and a Debug Adapter Protocol implementation. These come as two discrete sets of builtins (groups of commands), ‘dev’ and ‘spreadsheet’ with a surprise interaction or two.

The intent for the dev builtin is to eventually collect more and more useful tools for managing all aspects of software development, from source control management to building, testing and fuzzing.

Starting with the spreadsheet. By typing:

```
builtin spreadsheet
new
```

You would get something like the following screenshot:

#0 -1 No Data spreadsheet repeat X

[15:34:07][0][spreadsheet] > void#

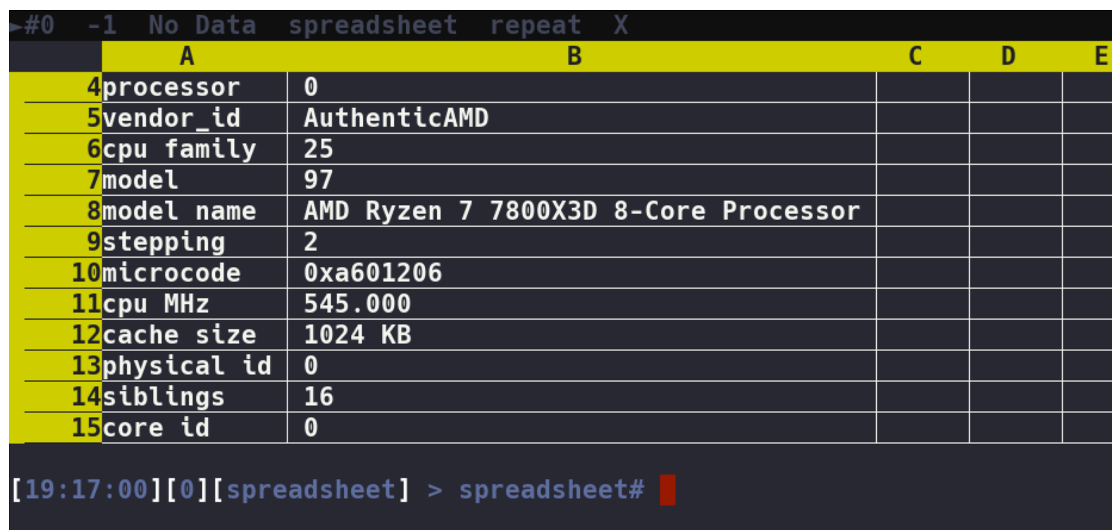
The following clip shows how I spawn new spreadsheet, populated by a CSV source and using the mouse cursor to interact with the layout.

This complements the form intended for media processing that was presented a few years ago as “Pipeworld” and re-uses the same language and parsing logic.

Cells can be populated with static contents, expressions like `=max(A1:A4)` or even shell commands through the ! prefix, e.g. `!date +%Y:%M:%S`. The following clip shows some basic expression use, as well as forced reprocessing of expressions and shell commands.

Combining shell commands with expressions that are re-executed on request

More useful is to populate the sheet with outputs from some command and processed by Lua patterns. The following screenshot shows the result of running `insert #0 4 separate "%s+:" !cat /proc/cpuinfo`



	A	B	C	D	E
4	processor	0			
5	vendor_id	AuthenticAMD			
6	cpu family	25			
7	model	97			
8	model name	AMD Ryzen 7 7800X3D 8-Core Processor			
9	stepping	2			
10	microcode	0xa601206			
11	cpu MHz	545.000			
12	cache size	1024 KB			
13	physical id	0			
14	siblings	16			
15	core id	0			

Populating a spreadsheet at some insertion point using a shell command split and separated with Lua patterns

Exporting can be done using the existing `copy` builtin with control over output format, subrange and so on: copy `#4(csv, compact, a1:b5)` to export the resolved values of a1,b1,a2,b2 … as CSV.

There is still more to be done for this to be a complete replacement for the likes of [sc-im,](sc-im) and to add things like plotting via gnuplot (if I can ever get their plot language to behave) and graphviz, as well more experimental things like importing Makefiles – — but at least for my, admittedly humble, spreadsheet uses it is good enough for daily driving.

Onwards to the Debugger

To start things you would do something like the following:

```
builtin dev
```

```
debug launch ./test
```

The ‘dev’ builtin set is used as it will eventually accumulate all developer task related commands like building, deployment, source control management and so on.

The following clip shows the default behaviour for that in action. In it you can see how multiple discrete jobs are created for managing threads, breakpoints and so on, detachable and with mouse cursor handling in place.

Basic debugger use and navigation

In the clip I also showed the process of stepping through threads, spawning source view, setting and toggling breakpoints, inspecting registers and modifying target variables.

I have spent thousands of hours staring at the GDB CLI prompt, and hated nearly every second of it. Not because of the relaxing task of debugging code or exploring software state itself, but for the absolutely atrocious terminal thrashing interface even with mitigation layers such as pwndbg. In fairness, a debugger TUI is in the deepest end of the complexity pool to get going.

There is a lot that goes into handling the protocol, and quite a few telltale signs of its designers, so we have just passed the point of basic bring-up. Importantly it is all composable with the data manipulation, filtering and transfer tools we already have elsewhere.

As an example of that we have ‘contain’ from the previous article, for instance, to bunch all the subwindows together into one contained job, useful when running multiple debug sessions side by side to step through multi-process issues.

We do have some other conveniences in place. Stepping controls are defined by the granularity of the job it represents, so stepping in the disassembly view would step instructions, while stepping in the source view would go by line or statement and so on.

Now to close the loop and mix in the spreadsheet part. In the following clip you see me picking a few registers and thread source location that gets added to a watch set. Whenever thread execution is stopped, these will be resampled and updated.

I then click the ‘spreadsheet’ option which will create a spreadsheet and populate it with the contents of the watchset as I go.

Live mapping watched dataset to spreadsheet

With all this in place we can almost start stitching the many other related projects together, from the data visualization from Senseye (closing in on 10 years…) with the window management from Pipeworld to the harnessing and setup from Leveraging the Display Server to Improve Debugging and build the panopticon of debugging from the plan presented in "Retooling and Securing Systemic Debugging" (2012, doi:10.1007/978-3-642-34210-3_10). But that is for another time.

About bjornstahl

-

View all posts by bjornstahl →

This entry was posted in Uncategorized. Bookmark the permalink.

1 Response to *A Spreadsheet and a Debugger walk into a Shell*

Peter Boos *says:*

September 17, 2024 at 13:17

It reminds of something when i was aged 14, i wrote out the assembler code of an MSX on raster paper, that way i could read and write assembly code

Reply