Making Trouble

Paul R. Potts

31 Dec 2021

Note: this week's issue includes excerpts of GNU Make files. The syntax highlighting won't appear in the version send via TinyLetter, but the standalone HTML file should include it. If the code is too wide to fit in your window, look for a horizontal scroll bar at the bottom of the code sections.

Thursday

It's been quite some time since I last sent out a newsletter. It's New Year's Eve Eve. In about 30 hours, it will be 2021. If I'm going There are plenty of reasons including losing a job, starting a new, and challenging job, and the stress of living through another year with the risk of COVID affecting everything we do. But I figure the way to get back into writing is to start. Maybe I'll be able to fill in the blanks later. And what's on my mind right now is a discussion of some new I.T. infrastructure at home, including a new server and some new computers, and what we're doing with this new infrastructure. This includes a new workflow for writing.

The Synology NAS

Recently I purchased a Diskstation DS620slim from Synology. This is a small server that holds six 2.5-inch hard disk drives or SSDs. I've long wondered if it would be possible to set up a home RAID server. I've periodically looked at available products, and they've always seemed a bit too complicated, power-hungry, noisy, and expensive. But this year when I looked at available servers, I came across this one, and it seemed to be just what I wanted. In particular it is small, quiet, and consumes very little power. It will sit on a shelf in my home office, attached to a UPS, and monitors the UPS status via a USB cable. If the UPS indicates the power is out, after a few minutes the server will shut itself down, well before the UPS runs out of power.

There's a very nice web interface. Since it has no screen or connector to plug in a monitor, the web interface is the only interface. Via another computer on the network, I was able to configure multiple accounts, privileged and unprivileged. The device allowed me to set up two-factor authentication. So to sign in with administrator privileges, I use a pass code delivered via an app on my phone.

Things got slightly confused with my initial order and the server came with six 3.5-inch hard drives instead of 2.5-inch hard drives, which left me looking glumly for a while at the empty server. I thought about returning all six 3.5-inch drives, but I had use for a couple of them as new backup drives. I might return four of them, or I might keep at least some of them on the shelf for future contingencies. I'll figure that out in the new year.

SSDs make ideal drives for this server, and enterprise-grade hard disc drives drives are also recommended, but it turns out that due to a new variety of cryptocurrency, there's a shortage of high-capacity 2.5-inch enterprise-grade hard-disc drives, and high-capacity 2.5-inch SSDs are extremely expensive at the moment. Cheaper drives are not recommended, but they were available, so I bought eight inexpensive 1-terabyte hard disc drives drives made for laptop use (six to fill the slots, and a couple of backups).

Configured with these six 1-terabyte drives in the recommended SHR-2 RAID configuration, using the **btrfs** file system, there are about 3.7 terabytes of storage available on the server. The rest is overhead used for data protection. Theoretically, the data is protected not only if one drive fails, but also if two drives fail. With two spares on the shelf, the idea is that I can swap out a bad drive or even two bad drives, and the RAID volume will be rebuilt without losing anything. The server will just run slower for a while as it reconstructs the contents of the drive. I am betting that at least four of the 1-terabyte drives will last for a year, but I will hedge that bet by purchasing more drives in the first quarter of 2022, and I will also make sure to get a backup strategy in place.

Upgrade Paths

Supposedly it's possible to not only replace individual drives, but to incrementally upgrade the storage capacity, by swapping in bigger drives. Because of the way data is duplicated across drives, though, swapping in just one higher-capacity drive won't upgrade the storage space. The available capacity wouldn't budge until I have upgraded four or more of the six drives.

Using the same SHR-2 RAID configuration, the numbers work out something like this for upgrading 4 of the drives:

Drive 1	Drive 2	Drive 3	Drive 4	Drive 5	Drive 6	Available Space
1TB	1TB	1TB	1TB	1TB	1TB	$<$ 4TB
2TB	2TB	2TB	2TB	1TB	1TB	$<$ 6TB
3TB	3TB	3TB	3TB	1TB	1TB	$<$ 11TB
4TB	4TB	4TB	4TB	1TB	1TB	$<$ 14TB

And like this for upgrading all the drives:

Drive 1	Drive 2	Drive 3	Drive 4	Drive 5	Drive 6	Available Space
2TB	2TB	2TB	2TB	2TB	2TB	$<$ 8TB
3TB	3TB	3TB	3TB	3TB	3TB	$<$ 12TB
4TB	4TB	4TB	4TB	4TB	4TB	$<$ 16TB

To put these numbers in perspective, my venerable Mac Pro (turned on in 2008 and almost never turned off since) currently has three drives in it: a 2-terabyte SSD for a system volume, which is a bit more than half full, a 4-terabyte spinning drive for media, which is almost full, and a second 4-terabyte drive for media, which is almost empty. The almost-full drive contains my complete iTunes library, containing lossless files imported from hundreds of compact discs. That amounts to almost a terabyte. It also contains the audio and video files for hundreds of personal projects (very space-intensive). By comparison, our family photo library and all spreadsheets, word processing files, and related documents represent only a small fraction of the space used; text files are tiny compared to video files.

If I upgraded the server to 8TB, I could easily use it to hold everything I've got and everything I'll be working on for the forseeable future, and stop using spinning discs for cold storage in the Mac Pro, instead just keeping current work on the SSD. That would be faster, too. But the future isn't easily forseeable. The kids are starting to become interested in 2-D and 3-D animation and audio and video production. If they really get going on that, they'll be creating a lot of large files. The server is here to enable them to learn and have fun like I have, so I'd better plan for more than 8 terabytes of capacity, although I won't need to add it all at once. I should plan on upgrading the server to use six 4-terabyte SSDs, although I have some time to watch for prices to come down.

The Backup Problem

The problem with these big volumes of data is backup. For each of these hard drives, I use two separate backup drives. When we lived in Saginaw, our bank had a very nice local branch housed in a beautiful old building, and I would periodically take one set of backup drives and put them in my safety-deposit box, and bring the other one home. This solved the off-site backup problem, which is really the "what happens if the house burns down?" problem. I don't have a great solution now. The server should be reliable, and survive some drive failures, but it still needs to be backed up. Synology has a cloud backup service, and I might wind up trying it.

Exceeding My Expectations

This thing has some features that have far exceeded my expectations. I've described the web interface for administration and the two-factor authentication.

But the NAS also supports two plug-in applications that solve a long-standing problem: sharing the music library to a network containing many kinds of clients.

iTunes supports a protocol for sharing a local library with multiple computers on the local network, as long as they are also running iTunes or Apple Music. I first tried to figure out how to use this sharing years ago: I wanted to maintain a music library in one place, and use a laptop to listen to it, pulling the music from the server and sending it to a set of speakers attached to an AirPort Express device. This never really worked at the time. I could get a little music out of it, but the WiFi throughput a dozen years ago or more was just not sufficient, and the playback would stutter terribly and freeze up while iTunes buffered more audio.

With newer WiFi routers, it works fine, and so we've been able to use a Mac Mini in the family room or a laptop running Windows and iTunes in the upstairs bedroom to play music from my Mac Pro. But I was never really happy with using the Mac Pro as a server. It has a feature which will keep it awake when there is network traffic over the wired network, but not when there is traffic over WiFi. If the Mac Pro is on WiFi, I have to set it to never sleep, which wastes a lot of power.

Because Apple no longer supports iTunes on newer operating systems, and newer Mac hardware requires newer operating systems, when I need to retire the Mac Pro, I won't be able to use iTunes as a server anymore. At some point the compatibilty problems introduced by maintaining very old operating systems and programs make it not worth the effort, so if I want to replace my Mac Pro with another Mac — and I'll still need a Mac if I want to be able to access any of my Logic Pro projects — I won't want to replace it with a machine that's old enough to still run iTunes. I'll want a Mac that can run modern software. And that Mac won't run iTunes.

I've also long wanted to find a good replacement for using iTunes as a *client*, so that I could stream music from my iTunes library to computers running Linux, or to phones. Unfortunately Apple's protocol includes a proprietary encryption key, so I wasn't able to find a good solution to this. Open-source clients weren't able to access files from iTunes acting as a server. And there aren't any closed-source clients other than Apple Music, which won't access iTunes server content on an iPhone, and which, I assume, will eventually be updated to phase out support for accessing iTunes server content entirely (Apple really would rather sell you their propriety cloud-based music sharing service, and I'm entirely uninterested in that).

The Synology NAS solves the obsolete server problem, the obsolete client problem, and the mixed client problem.

There's a plug-in which will serve music directly from the server to any copies of iTunes running on the local network. Music playback was a bit irregular and stuttered while the server was initially building the volumes, but once that was done, it works just fine.

There's *also* a plugin called Audio Station and a corresponding app called DS Audio for iPhone and Android, which completely solves the problem of accessing music files from the server from a phone on the local network.

And for any other computer with a web browser, they can simply connect to the server, using an unprivileged account that has read-only access to the music library, and run a web-based application which looks an awful lot like iTunes.

To put new music into the library, I can still import it into iTunes on my Mac Pro, and then use the command-line tool **rsync** to update the files on the server. I think I need to tell the server to manually update the index, but that's not really a problem. iTunes isn't really required; it's just a convenient tool that I've been using for a long time. But Apple doesn't support it on newer computers anyway, so I will eventually phase it out.

Since most of the music in the library is in Apple Lossless format (aka ALAC), I was concerned that the clients wouldn't all be able to play these files. I wasn't looking forward to converting all the files to WAVE format, which would involve doubling their sizes and losing a lot of the MP3 tags. But I needn't have worried. All the files play very nicely via web browser, via app, and via iTunes. I think iTunes for Windows is lacking some codecs, but we've currently got only one laptop running Windows. We keep it around because Zoom never worked right on the Linux laptops. We're planning to phase out Windows altogether in 2022.

I'm not quite ready to phase out macOS entirely, but we are putting increasing effort into using Linux for more and more. Which brings me to the next topic.

Central Dogma

The new NAS server is only a part of our strategy to upgrade our I.T. infrastructure. In order to give the kids more options for computing, and give a few to the adults as well, I bought three new PCs (sort of).

The network device name of the new server is **central-dogma** taken from the animated series Neon Genesis Evangelion. The music library is called **nerv_music**, which is a reference to the NERV organization in the Evangelion universe, but also a reference to Laurie Anderson's "Nerve Bible." In Central Dogma, deep underground in NERV, there are three giant artificial intelligences, called Balthasar, Melchior, and Casper.

Our computers called **balthasar**, **melchior**, and **casper** aren't three giant artifically intelligences. A few weeks ago I found that B&H Photo Video had three small computers in stock, Intel NUCs ("Next Unit of Computing") of the type known as "Bean Canyon." These are tiny boxes, smaller than a Mac Mini, but they have i5 processors in them. I chose these because they were inexpensive and also because they are very low-power; one of them is sitting on the shelf downstairs next to the NAS server, for remote use by the kids who are learning programming. They can SSH into it, or they can use the desktop remotely using

the RDP protocol. They can mount their own network volumes from the server and keep their files there, accessible from any machine on the network.

These little computers came without any drives and without any operating systems. I ordered memory and SSDs. The boys and I installed the memory and SSDs — this was very easy to do, much easier than building a regular PC from parts — and closed them up. I installed Ubuntu Server to start with. There were a few issues I came across using Ubuntu Server. For example, by default, the devices would go to sleep after a little while, so I had to figure out a fix for that. It's a strange default configuration for a headless server machine. But once I found the workaround, the fix was easy.

I thought at first we might use these boxes entirely via the command line, but although I'm pretty good with command-line tools, I'm just too accustomed to doing some things with GUIs. So, I installed the light-weight **lightdm** window manager and the standard Ubuntu desktop environment. While I was at it, I replaced the Ubuntu MATE desktop environment on the old HP laptops, to make the machines more consistent. I never really liked the MATE desktop anyway. The laptops and the mini-PCs are all running Firefox. The only issue I've run into is that Paramount Plus won't play streaming video with Firefox. So I'm using Chrome for that.

Oh, I almost forgot — there aren't really three of them yet. There are only two. Casper went missing. Apparently, B&H didn't really have three of the mini-PCs in stock. So I got the memory and SSD, but only two of the computers. After a few days, I got a refund for the price of the third computer. These little NUCs are very popular and supplies are constrained. This Bean Canyon i5 is out of production now. I've looked to see if I can find one more of this exact same model, a new-old stock or used box, but the units I've found in stock are much more expensive than the ones I purchased. Fortunately there are newer models available for about the same price. The 2400-speed DDR4 DIMMs I already bought should work with these newer devices, although since the newer devices support slightly faster memory, they will operate a little bit slower than their maximum memory bandwidth. I'll either live with that, or return the older memory and order some new DIMMs that match the maximum speed of the new device.

The kids want to try running games on these little boxes, and it's possible, at least at reduced resolution, but I don't want to encourage the kids to use them for gaming at present. I don't think it's a great idea to run these tiny boxes under that kind of heavy load — it will probably shorten their lifespan. Also, PC gaming can quickly become an all-consuming and very expensive hobby. If high-end video cards are ever available again and the older kids want to save up their allowance and build a gaming PC from parts, I'll gladly help them build it, but as a separate project.

If you are considering a Mac Mini or other PC for general use (not gaming), I highly recommend getting one of these NUCs and installing your own memory

and SSD and installing the regular version of Ubuntu. It makes a very useful, tiny machine. It is even possible to mount them on the back of certain monitors to make a sort of poor man's iMac, but it's hardly poor; these are quite powerful little boxes and will play streaming video content without stuttering. I'm currently using one of them in our bedroom on a 43-inch 4K monitor to do some writing and programming and it is impressive. I've had no issues at all other than the minor hiccups in configuration that I mentioned.

Next up, I'm going to describe the new workflow that the combination of NAS server and mini-PCs has allowed and encouraged me to develop.

The Mac-Free Workflow

I've used plenty of different kinds of computers in the last 45 years, but for actually getting writing done, I've long preferred Macs. My favorite editor is BBEdit. It's a reliable workhorse and very fast. I'd rather use BBEdit than any other editor. BBEdit is Mac-only, though. Notepad++ is a pretty capable alternative on Windows, and if I have to work on a Windows machine, and if I'm allowed to allow software on it, I quickly install Notepad++. Why? The have features that I use all the time — for example, both of them allow rectangular selection. But I haven't truly ever found a similar text editor that I loved on Linux. I can do the basics in **vi**, but I'm just not as quick and effective when writing in **vi** as I am using a visual editor.

I am still getting used to it, and I can't say I love it, but Visual Studio Code is pretty decent. I love the themes — they work very well on modern high-dynamic-range screens where pure white or pure black are just too bright or too dark and the extreme contrast is hard on the eyes. I'm partial to the "Monokai" family of themes.

The Mac version of Visual Studio Code feels speedier and seems to be ahead of the Linux version, but the Linux version is OK, too. But one thing I really miss is BBEdit's worksheet concept, borrowed from MPW, the old Macintosh Programmer's Workshop. Worksheets are editable documents which allow you to select text and execute it. The output of the commands shows up right in the worksheet. So it's like using the command line, except tht you can visually edit the commands you are issuing, and keep a log of the results. I use worksheets extensively for producing podcasts and for writing. When I start a new podcast, I will often copy and paste the text I used for a previous episode, update the title and some details, and go from there. I do similar things with the sets of commands I use to generate HTML and PDF files from Markdown files.

Visual Studio Code doesn't have worksheets, but it does allow me to open up a convenient terminal pane right in the GUI. I've long wondered how hard it would be to use **make** for my writing workflow. With a little time off to work on it, I decided to dive and try it.

The Basics of make

The **make** utility program, which on Linux usually means GNU Make, but can mean other similar tools on other platforms, is a very old program designed in a different computing world. I've used many versions of **make** over the years on many platforms, going back to MS-DOS, and I understand the basic idea. I've written some simple Makefiles, but I never really dove into more complicated uses. Searching the web, I found some blog posts and Github projects that include Makefiles for working with **pandoc**, but I was not able to easily understand them, and they seemed to lack some of the features I needed. I spent a little time looking for a more modern alternative to **make** that would be suitable, but they all seemed too heavyweight, too language-specific, or too complex.

I own a print copy of the GNU Make manual written by Richard M. Stallman and Roland McGrath, and that seemed like a good place to start. Unfortunately, it really wasn't. Although the book does have some Makefile examples into it, it is mostly an in-depth reference to the more arcane built-in functions. The examples included are extremely simple and don't use most of the features described. So, I had to try to learn the hard way, building a Makefile from the ground up, and simply experiment until I figured some things out.

When I say that **make** "is a very old program designed in a different computing world," here are a few examples of what I mean by that:

- It operates on directories and files, but the lists of files are handled internally as text strings.
- Make doesn't have anything like modern quoted string types, with support for escaped characters.
- Make's support for wildcards doesn't look much like it does in other programming environments you might be used to, such as Bash or Perl.
- Make offers library functions for pattern-matching rather than regular expressions.
- It operates on lists of "words" (delimited by spaces).
- The use of spaces as word delimiters results in strict limitations on the characters that can be in directory and file names.
- Make does not have modern facilities for working with directories, other than as strings.

And, finally,

- Commands in rules must always be indented using tab characters, not spaces!

That last thing is a minor irritant, but it's one of the longest-lasting minor irritants in the entire history of software development. Just in case you wind up trying to copy and paste excerpts from my Makefile, below, into one of your own, you should be aware that, depending on which tools you are using, the resulting text may not contain the necessary tab characters, and if it doesn't, **make** will complain about that.

My make Use Case

What I'm trying to do doesn't seem too hard. I'm working with directories full of Markdown files that are part of project directories. The source tree is also version-controlled using Git. The structure looks something like this:

```
[my personal server directory]/
    writing/
        src/
            the_coffee_underachiever/
                Makefile
                md/
                    2019/
                        file1.md
                        file2.md
                        file3.md
                    2020/
                        file4.md
                        file5.md
                    2021/
                        file6.md
                img/
                index.md
```

I want this to get transformed into a somewhat different structure, where the generated files are staged to be synchronized with my web host:

```
[work dir]/
    sites/
        writing/
            the_coffee_underachiever/
                2019/
                    file1.html
                    file2.html
                    file3.html
                    pdf/
                        file1.pdf
                        file2.pdf
                        file3.pdf
                2020/
                    file4.html
                    file4.html
                    pdf/
                        file4.pdf
                        file5.pdf
                2021/
                    file6.html
```

```
        pdf/
            file6.pdf
    index.html
    img/
```

Defining My Variables

The **make** program allows a Makefile to define variables. They are really space-delimited strings. We specify directories including the trailing slash.

```
SRC_ROOT_DIR = .
MD_SRC_ROOT_DIR = $(SRC_ROOT_DIR)/md/
IMG_SRC_ROOT_DIR = $(SRC_ROOT_DIR)/img/
DEST_ROOT_DIR = ~/Documents/sites/writing/the_coffee_underachiever/
DEST_PDF_SUBDIR = pdf/
DEST_IMG_DIR = $(DEST_ROOT_DIR)img/
```

For example, **DEST_IMG_DIR** becomes "~/Documents/sites/writing/the_coffee_underachiever/img/"

We can do wildcard expansion in many places in Make, but the * syntax doesn't work when defining a variable; instead we have to use the built-in **wildcard** function. To include the contents of a variable, we use the **$(variable_name)** syntax. We can combine these and define a variable that contains a list if space-delimited "words," where each "word" will be a filename with its path, for example **./md/2019/file1.md**.

```
SRC_MD_DOCS = \
    $(wildcard $(MD_SRC_ROOT_DIR)*.md) \
    $(wildcard $(MD_SRC_ROOT_DIR)2019/*.md) \
    $(wildcard $(MD_SRC_ROOT_DIR)2020/*.md) \
    $(wildcard $(MD_SRC_ROOT_DIR)2021/*.md)
```

I'd like it if there was a **make** function which would search a subtree starting from a given point — for example, I'd like it if I could use a single built-in function to create a list of full paths to all the **.md** files in the **md** directory and its subdirectories, recursively, but I don't think there is. I think it's possible to do this by including shell commands in the Makefile itself. I've seen examples of this, but for various reasons I'm not happy with this technique, so I'm going to avoid it for now.

Note that the "list" generated by the variable definition is really not even a list in the Lisp sense, but a single string consisting of space-delimited "words." The first few words are:

```
./md/index.md ./md/2019/file1.md ./md/2019/file2.md
```

Note that this means the filenames and directory names can't contain any spaces! This was normal in the days when systems ran early versions of UNIX, as well as CP/M and MS-DOS. But severe restrictions like this haven't been common since

the development of more user-friendly systems, which include modern versions of UNIX. So we now have a tool that runs on UNIX and Linux systems that imposes much more severe restrictions on filenames than the systems themselves.

I don't like this restriction; many of my files contain spaces as well as other characters that cause trouble with tools such as **make**, including single and double quotation marks. I'd rather work with a tool with facilities that will handle arbitrary filenames and directory names, but for now I'm willing to live with these limits and change my filenames to conform to these requirements. Long-term, I'll be looking for a more modern tool.

Creating My Target File Lists

Anyway, I've now got a list of all the Markdown files, but to specify **make** rules for creating targets from the prerequisites, I need to specify the targets. The **make** utility provides a number of functions that process these lists of files. As I mentioned before, the manual I was working from is very light on real-world examples, so I had to experiment. Here's a variable definition I came up with for generating a list of my target HTML files from the list of precedent Markdown files. I'll present the definition first and then explain it a bit.

```
DEST_HTML_DOCS = \
    $(subst $(MD_SRC_ROOT_DIR),$(DEST_ROOT_DIR), \
        $(join \
            $(dir $(SRC_MD_DOCS)), \
            $(addsuffix .html, \
                $(basename $(notdir $(SRC_MD_DOCS))))))
```

The multiple levels of indentation are not strictly required, but I wrote it this way because to me, the Make functions look a bit like Lisp primitives, which makes sense, given Stallman's background and work on Emacs Lisp. Working from the inside out, we start by applying two functions to **$(SRC_MD_DOCS)**, **dir** and **notdir**. The **dir** function takes a word, or list of words, and returns only the part or parts that look like directory paths, not filenames, using a simple heuristic (recall our restrictions on directory names). The **notdir** function gives us only the filenames. So we've got two "lists" now:

```
./md/ ./md/2019/ ./2019/
index.md filename1.md filename2.md
```

I apply the **basename** function to the output of **notdir**, which yields the filenames without extensions:

```
index filename1 filename2
```

And then the **addsuffix** command:

```
index.html filename1.html filename2.html
```

The **join** command is, if you squint, a bit like a list **zip** command in a language

like Haskell. It assembles the elements from two lists of the same length into one, giving us:

```
./md/index.html ./md/2019/filename1.html ./md/2019/filename2.html
```

Finally, we use the **subst** function to replace a substring of each word, giving us:

```
~/Documents/sites/writing/the_coffee_underachiever/index.html ~/Documents/sites/writing/the_
```

If it's occurred to you that using textual substitution on filename paths is fragile and subject to all kinds of breakage, especially since it doesn't appear that this substitution will only take place starting from the beginning of the string, you're absolutely right! There is certainly a better way, but probably not one that can be implemented entirely in **make** — it's built on very soft foundations more suited for a simpler and gentler computing ecosystem.

For generating the paths for the PDF files, I have a similar definition, except that I don't want to generate a PDF file of the index. So I filter that word out:

```
SRC_MD_DOCS_NO_INDEX = \
    $(filter-out $(MD_SRC_ROOT_DIR)index.md, $(SRC_MD_DOCS))
```

Then I generate the list of PDF targets like I did the HTML targets:

```
DEST_PDF_DOCS = \
    $(subst $(MD_SRC_ROOT_DIR),$(DEST_ROOT_DIR), \
        $(join \
            $(dir $(SRC_MD_DOCS_NO_INDEX)), \
            $(addprefix pdf/, \
                $(addsuffix .pdf, \
                    $(basename $(notdir $(SRC_MD_DOCS_NO_INDEX)))))))
```

This definition is obviously very similar to the previous one, differing only by the prefix and suffix. There may be a way to factor out a common function here, but I'm not sure; while **make** is, I think, certainly Turing-complete, it is lacking a lot of things that I think of as fundamental to programming languages.

Generating Targets with Pandoc

Here's some scaffolding for generating **pandoc** commands:

```
PANDOC=/usr/bin/pandoc
PANDOC_OPTIONS=--ascii --standalone --shift-heading-level-by=-1 \
    -f markdown+smart
PANDOC_HTML_OPTIONS=--to html5
PANDOC_PDF_OPTIONS=
```

Now that we've got our lists of documents, we can write rules that match on them. Let's write a rule that should turn any of our Markdown files into corresponding HTML files, where the % character matches any substring in our file path — but note that % must match the **same thing** on both sides.

```
$(DEST_ROOT_DIR)%.html : $(MD_SRC_ROOT_DIR)%.md
    $(PANDOC) $(PANDOC_OPTIONS) $(PANDOC_HTML_OPTIONS) -o $@ $<
```

In this case it will match on a substring like:

`2021/2021_02_21_Shelving_the_Library`

which is found in both the target:

`/home/paul/Documents/sites/writing/the_coffee_underachiever/2021/2021_02_21_Shelving_the_Lib`

and the predecessor:

`md/2021/2021_02_21_Shelving_the_Library.md`

But the following rule will **not** work for PDF files, since the **pdf/** subdirectories exist in the output directories:

```
$(DEST_ROOT_DIR)%.pdf : $(MD_SRC_ROOT_DIR)%.md
    $(PANDOC) $(PANDOC_OPTIONS) $(PANDOC_PDF_OPTIONS) -o $@ $<
```

Instead we need to match on a rule that takes the structural difference into account. We can't just append the **pdf/** subdirectory on the left, since the match includes the filename, so this won't work:

```
$(DEST_ROOT_DIR)%pdf/.pdf : $(MD_SRC_ROOT_DIR)%.md
    $(PANDOC) $(PANDOC_OPTIONS) $(PANDOC_PDF_OPTIONS) -o $@ $<
```

Have I mentioned that debugging Makefiles can be quite difficult? Well, it can be!

And unfortunately our pattern matching options seem to be limited; we don't have regular expressions in our toolkit. So I had to use multiple pattern-match rules for my PDF file outputs, where the % matches only on the filename portion of the file path:

```
$(DEST_ROOT_DIR)2019/pdf/%.pdf : $(MD_SRC_ROOT_DIR)2019/%.md
    $(PANDOC) $(PANDOC_OPTIONS) $(PANDOC_PDF_OPTIONS) -o $@ $<

$(DEST_ROOT_DIR)2020/pdf/%.pdf : $(MD_SRC_ROOT_DIR)2020/%.md
    $(PANDOC) $(PANDOC_OPTIONS) $(PANDOC_PDF_OPTIONS) -o $@ $<

$(DEST_ROOT_DIR)2021/pdf/%.pdf : $(MD_SRC_ROOT_DIR)2021/%.md
    $(PANDOC) $(PANDOC_OPTIONS) $(PANDOC_PDF_OPTIONS) -o $@ $<
```

There may be a more concise way to handle this, but this is working fine for now.

Note that the special variables: **$@** and **$<** mean, respectively, the target and the predecessor, that matched the left and right side of the rules.

```
$(DEST_ROOT_DIR)index.html : $(MD_SRC_ROOT_DIR)index.md
    $(PANDOC) $(PANDOC_OPTIONS) $(PANDOC_HTML_OPTIONS) -o $@ $<
```

Then I can supply a pattern rule:

```
$(DEST_ROOT_DIR)%.html : $(MD_SRC_ROOT_DIR)%.md
    $(PANDOC) $(PANDOC_OPTIONS) $(PANDOC_HTML_OPTIONS) -o $@ $<
```

This rule will match on file paths that match the destination and source root directories with anything in between these directories and the file suffixes. The % character must match the same string of characters on both the left (target) and right (predecessor) side. So a target of:

`~/Documents/sites/writing/the_coffee_underachiever/2019/filename1.html`

and a predecessor of:

`./md/2019/filename1.md`

will result in the command under this rule being executed, to generate the target from the prececessor, and the whole point of **make** is to only run what needs to be re-run if a predecessor has been changed more recently than its associated target.

I haven't really shown how I handle the images, but essentially I just copy all the matching JPEG files from the source into the destination with a simple rule that uses the **cp** command:

```
$(DEST_IMG_DIR)%.jpg : $(IMG_SRC_ROOT_DIR)%.jpg
    cp $< $@
```

In the future I might create image subdirectories like I do with with my PDF subdirectories, but that shouldn't be hard to change.

Defining Makefile Goals

Now, here's how to define goals. The rules are a bit obscure here. Invoking **make** with no goal will not always do what you want, so we need to define an **all** target and make it so that the default is set to match this target. I also know that it takes a very long time to generate all the PDF files, so I don't want the usual **make clean** command to remove them. I want to make it so **make pdfclean** removes them instead. That will help make it so I don't accidentally make it so I have to regenerate all the PDFs, which might require a half-hour or more, unless I've changed a predecessor.

The **.PHONY** goal is slightly difficult to explain, but essentially, the parameters to **make** can be used to specify a target filename or filenames. This works fine unless we have a filename that conflicts with our goal. To avoid this possibility, we declare these targets as **.PHONY**, meaning that they define goals and not actual target files. This elaborate workaround wouldn't have been necessary if **make** supported a more modern set of command-line options, like **make –goal=html** or **make –target=index.html**.

```
.PHONY: all imagefiles htmlfiles pdffiles \
    imageclean htmlclean pdfclean clean

.DEFAULT_GOAL := all

all: imagefiles htmlfiles pdffiles

imagefiles: $(DEST_IMG_DOCS)

htmlfiles: $(DEST_HTML_DOCS)

pdffiles: $(DEST_PDF_DOCS)

imageclean:
    rm $(DEST_IMG_DOCS)

htmlclean:
    rm $(DEST_HTML_DOCS)

pdfclean:
    rm $(DEST_PDF_DOCS)

clean: imageclean htmlclean
```

The goals can consist of both targets and commands. Building the PDF target processes specifies the PDF targets. Cleaning the PDF targets does not attempt to build any targets, but executes the **rm** command on all the PDF targets. Goals can refer to other goals.

Creating Directories: the Simple Way

I've left off a useful step. Given what I've shown you so far, you'd have to manually create the target subdirectories. We can use **make** to do that for us, but doing it the right way complicates the Makefile a bit.

We could just add some **mkdir** commands to our existing goals, using the **-p** option, which makes it so no error is generated if the directory already exists:

```
html: $(DEST_HTML_DOCS)
    mkdir -p $(DEST_ROOT_DIR)2019/
    mkdir -p $(DEST_ROOT_DIR)2020/
    mkdir -p $(DEST_ROOT_DIR)2021/
```

This requires some overhead every time **make** is run with this goal. It's pretty insignificant in this case, but to do it the right way, so that it doesn't send unnecessary commands, you use "order-only prerequisites."

Creating Directories: the Idiomatic Way

Let's add a rule to indicate that the image files targets are dependent on the image directory:

```
$(DEST_IMAGE_FILES): | $(DEST_IMAGE_DIR)
```

Note that there's some new syntax here. Prerequisites mentioned in a rule to the right of a vertical bar character are "order-only prerequisites." They are checked to see if they exist, but their time stamps are not checked to determine if they are newer than the target. It makes sense to use this kind of prerequisite for directories, whose time stamps are updated whenever something in the directory changes. If we didn't do this, we could get rules re-triggered when both a file time stamp and its containing directory time stamp are updated.

Now we have a rule to make the image directory. Since it only runs if the order-only prerequisite image directory is missing, we don't need to supply the **p** option to **mkdir**.

```
$(DEST_IMAGE_DIR):
    mkdir $(DEST_IMAGE_DIR)
```

The **imagefiles** goal just expands to all the image file targets:

```
imagefiles: $(DEST_IMAGE_FILES)
```

Now we have similar sets of prerequisites for our HTML output directories. The order-only prerequisites are expanded from a variable **DEST_HTML_DIRS** that I've defined to contain the three destination directories.

```
$(DEST_ROOT_DIR)2019/:
    mkdir $(DEST_ROOT_DIR)2019/

$(DEST_ROOT_DIR)2020/:
    mkdir $(DEST_ROOT_DIR)2020/

$(DEST_ROOT_DIR)2021/:
    mkdir $(DEST_ROOT_DIR)2021/

$(DEST_HTML_FILES): | $(DEST_HTML_DIRS)

htmlfiles: $(DEST_HTML_FILES)
```

Things are a bit more complex for the PDF files, since the PDF subdirectories exist inside the HTML directories. This means I want to specify that making the destination PDF files depends on the destination PDF directories, and also specify that making the PDF directories depends on making the HTML directories. If there's nothing in the destination root directory, when I run **make pdffiles**, these rules will result in **make** creating the HTML directories first, then the PDF subdirectories, then the PDF files.

Note that if the HTML directories have already been created, **make** will only create the PDF subdirectories, then the PDF files. If both sets of directories exist, then **make** will only create the PDF files themselves.

Why are there three targets for the PDF subdirectories? Well, that makes it easier to reliably handle cases where one or two of the three subdirectories is missing, without generating errors.

```
$(DEST_ROOT_DIR)2019/pdf/:
    mkdir $(DEST_ROOT_DIR)2019/pdf/

$(DEST_ROOT_DIR)2020/pdf/:
    mkdir $(DEST_ROOT_DIR)2020/pdf/

$(DEST_ROOT_DIR)2021/pdf/:
    mkdir $(DEST_ROOT_DIR)2021/pdf/

$(DEST_PDF_DIRS): | $(DEST_HTML_DIRS)

$(DEST_PDF_FILES): | $(DEST_PDF_DIRS)

pdffiles: $(DEST_PDF_FILES)
```

Cleaning Up Directories

That just leaves our goals to clean things up, which shouldn't be too hard to understand now. Since the Makefile will properly build any needed directories, the **imageclean** target just removes the whole image directory rather than just the files inside it:

```
imageclean:
    rm -rf $(DEST_IMAGE_DIR)
```

For the **htmlclean** target, I don't actually want to remove the directories, because they might contain PDF files. Rebuilding the PDF files is time-consuming, so we don't want to do it unless the **pdfclean** goal was specified.

```
htmlclean:
    rm $(DEST_HTML_FILES)
```

For the **pdfclean** target, we can remove the PDF subdirectories.

```
pdfclean:
    rm -rf $(DEST_ROOT_DIR)2019/pdf/ \
        $(DEST_ROOT_DIR)2020/pdf/ \
        $(DEST_ROOT_DIR)2021/pdf/
```

You might notice that the way I've implemented the clean commands means that I'll never actually delete the directories **2019**, **2020**, and **2021** under the destination root directory, once they've been made. There are ways to

conditionally remove directories only if they are empty, but they seem quite ugly to me and depend on shell commands, so I'm not going to bother with that. I can live with some leftover empty directories.

The New Workflow

So, now I've got a **Makefile**. Now what?

Well, this means that whenever I make changes in the source files, I can just execute **make** in the Visual Studio Code terminal window and it does only the minimal amount of work required to bring the targets up to date. The staging directory is local to the machine I'm working on — today it's a laptop, yesterday it was Melchior, one of the NUCs. I don't worry about backing up the contents of the staging directories, since all the files are generated from the version-controlled source files on the server.

Finally, when I am satisfied with the generated HTML and PDF files, I can push them up to our web server with **rsync**. I'm going to save that topic to explain another day, since it involves SSH keys.

Happy New Year!

About This Newsletter