How I Use GNU Make

Paul R. Potts

December 2021

Below I describe why and how I use GNU Make, together with other tools, to generate web pages and PDF files from my common source code in Markdown.

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.
