Pierce etc.

Make

Introduction

Published: 2015-04-02
Updated: 2015-04-02

`make` is a widely used and valuable development tool. It's a "build" tool: it builds programs and documentation according to a "recipe". It could really be used for anything where someone edits some files, and then runs a series of processing steps to generate some other form from the edited files. For the most part, however, it's just used to build and install software. `make` has its origins in Unix somewhere, and these days each BSD project and the GNU project have their own version.

I often get the impression that many otherwise knowledgeable and skilled developers don't have more than rudimentary knowledge of `make`, and could benefit from a more solid understanding. I don't particularly blame them: `make` is certainly ancient and has odd syntax and quirks. So many developers do the minimum necessary to add their new sources to the build, and then go back to working on the actual code. Having a good build system and understanding how it works can make development and deployment of software much more pleasant, so I humbly suggest taking the time to really learn one.

This blog post is about using a subset of the features of GNU Make to write "good" build systems. There are a variety of build systems out there, some of which use `make` as a component, like CMake and GNU Autotools, and some of which don't, like scons or jam or ninja. I won't fault anyone for using another build tool if it's convenient, but I will judge it based on whether it gets reasonably close to this simple goal:

A developer should be able to run a single command (like `make [args]`) and get the correct result ASAP.

In more detail:

- The build system should produce the exact same final outputs as if it was doing a completely clean from-scratch build, even when it's not.
- The build system should be flexible.
- The build system should be reasonably fast.

Flexibility:

- The build system should work with different filesystem hierarchies, where the locations of dependencies and outputs can be flexibly specified.
- The build system should be able to install to an arbitrary "root" directory, so the results can be easily packaged, or simply installed into a filesystem tree for a separate system.
- The build system should be able to be configured to do a cross-compile for a different architecture, if applicable. It's usually enough to enable the compiler and linker paths and options to be conveniently specified.

Speed:

- A fast build system will avoid unnecessary work, re-using existing results if they would be exactly the same if regenerated.
- A fast build system will do tasks in parallel when possible.

Developers of the software don't want to wait too long to see and test the result of a change. Neither do integrators working on an embedded OS or a large application, and for them, reducing a number of short build times can add up to a big difference. Those developers also need the flexibility to be able to integrate the

software into their system without too much patching.

If a developer is working on a piece of software, makes a change, runs a build, and later figures out that the change didn't get into the build, he'll become paranoid and always clean the source/build tree before a build. This really hurts a quick feedback cycle.

Why GNU Make

In this blog post I'm going to use GNU Make because:

- Almost every somewhat-popular OS has a convenient copy of GNU Make already installed or available from the OS distributor, including the BSDs, Mac OS X, Linux distros, and other UNIXes. Windows is, admittedly, a more complicated story, which I won't address here.
- The `make && sudo make install` convention is very common. If you've built more than a couple of packages of open source software you found on the web, you've probably already used `make`.
- If you need to debug a build system of which `make` is a component, perhaps related to trying to use some of the flexibility mentioned above and finding it slightly broken, it will help to have a solid understanding of the fundamentals of `make`.
- It's possible to build a correct, flexible, and fast build system with `make`, and keep it relatively simple.
- GNU Make only depends on the standard C library (and optionally "guile", but don't use that).

Of the various implementations of `make`, I've only used GNU Make, and even as I try to stick to a reasonable subset of the features of GNU Make, I'll use at least some GNU extensions which are not the same in the BSD versions, for example `$(patsubst ...)`. However, the basics are the same for the BSD make implementations, and this should still give you a good grasp of them, enabling you to debug and create decent `Makefile`s for any unix/make still in use.

A re-introduction to make

You might think of Makefiles like this (intentionally **bad** example):

```
# subroutine def  subroutine call   another subroutine call
#     v              v                v
install:   install-doc    build-binary
    install binary $(bindir)/
    mkdir -p $(datadir)/myprog/
    cp -r images $(datadir)/myprog/
#       ^
#       body of subroutine
```

That was a `make` *rule*, where the *target* is "install", the *prerequisites* (or *dependencies*) are "install-doc" and "build-binary", and the rest is the *recipe* - commands for the rule, indented with a tab. The tab is special. A single tab must be used to indent recipes, and nothing else.

This model obscures the property that "install-doc" and "build-binary" could happen in parallel with each other if `make` is invoked with "-j2". It also obscures some ways in which unnecessary work could be avoided, and that this rule might not run if a local file named "install" was accidentally created.

Makefiles should be more like a graph of dependencies than a script. Most nodes in the graph are files. Script snippets connect the nodes. I think of the whole graph as a structure that rests on the human-edited source files, and builds up with beams and joints to support the desired output files in the desired location. By changing a variable, the whole structure instantly reconfigures to support different outputs in a different location, re-using as much as possible of the existing parts of the structure. When this mechanism is robust and works well, I find it to be pretty cool.

I'm going to start by emphasizing the mechanics of the graph, and then I'll more quickly introduce a tasteful selection of features that are useful for conciseness and elegance. Also, at times I'll say "you can't do that" and omit "unless you use this other feature and this trick", for brevity. I urge you to minimize the use of "tricky" features, for both your own benefit and that of the next person who wants to modify the Makefile.

Every target is a file

Every target is a file, except for the ones which are PHONY, which should be the less common case. You can list phony targets as prerequisites of the `.PHONY` target, like so:

```
.PHONY: install all clean
```

This prevents `make` from being confused by a file with the same name as the phony target. In some cases it can also help clarify for someone reading the Makefile that the target is phony.

The rules for phony targets are always run, if the phony target is a prerequisite of some target that's needed. The rules for file targets, on the other hand, are only run if the file doesn't exist or is older than any of its prerequisites.

Let's use this little Makefile to experiment:

```
all: program
    @echo building all

empty-all: program

program: program.o extra.o
    gcc -o program program.o extra.o

program.o: program.c
    gcc -o program.o -c program.c

extra.o: extra.c
    gcc -o extra.o -c extra.c

clean:
    rm -f program program.o extra.o

.PHONY: all empty-all
```

Make prints out each command before it executes it, except for commands prepended with `@`, which I use to avoid printing the "echo" command itself before it prints something.

I'll demonstrate some of the properties of this Makefile:

```
$ make all
gcc -o program.o -c program.c
gcc -o extra.o -c extra.c
gcc -o program program.o extra.o
building all
$
$ # "program" is up-to-date, but "all" is phony
$ make program
make: `program' is up to date.
$ make all
building all
$
$ # you can specify multiple targets
$ make extra.o program.o
make: `extra.o' is up to date.
make: `program.o' is up to date.
$
$ # "all" is the default target, because it's the first target in the Makefile
$ make
building all
$
$ # watch an updated target cause other targets to update in a cascading manner
$ touch extra.c
$ make
gcc -o extra.o -c extra.c
gcc -o program program.o extra.o
building all
$
$ # forgot to mark "clean" as PHONY, but it works as long as a file named "clean" never exists
$ touch clean
```

```
$ make clean
make: `clean' is up to date.
$ rm clean
$ make clean
rm -f program program.o extra.o
$
$ # it's normal for "clean" to behave more like a script, and fully run if called again
$ make clean
rm -f program program.o extra.o
$
$ # compare "clean" to "all", which is properly marked PHONY
$ make all
gcc -o program.o -c program.c
gcc -o extra.o -c extra.c
gcc -o program program.o extra.o
building all
$ touch all
$ make all
building all
$
$ # compare "all" to "empty-all", which is also marked PHONY, but has no commands
$ make clean
rm -f program program.o extra.o
$ make empty-all
gcc -o program.o -c program.c
gcc -o extra.o -c extra.c
gcc -o program program.o extra.o
$ make empty-all
make: Nothing to be done for `empty-all'.
```

A target is "up to date" if the file with the name of the target exists and is newer than all its prerequisite files. Some other build tools calculate hashes of the file contents, and some also include build parameters in the hash, but `make` is all about file change timestamps. In the demonstration, "touch" is sufficient to update the timestamp and cause `make` to think the file has changed.

File targets should only depend on file targets

Because phony targets are never up to date, a phony target will cause any file targets that depend on it to be needlessly rebuilt (which will cascade up the dependency tree). Don't make any file target depend on a phony target.

Phony targets ideally have no commands

"clean" is the common exception to this rule, because the result of "clean" is not files, but the absence of files. "test" or "check" is another common exception. But for almost any other phony target rule, have it instead depend on the files it would create. I'll add an "install" target to the Makefile above which follows this rule:

```
install: /usr/local/bin/program

/usr/local/bin/program: program
    cp program /usr/local/bin/program

 # this is additive to the existing list of prerequisites for .PHONY
.PHONY: install
```

A quick demonstration:

```
$ make clean
rm -f program program.o extra.o
$ make install
gcc -o program.o -c program.c
gcc -o extra.o -c extra.c
gcc -o program program.o extra.o
cp program /home/plo/bin/program
$ make install
make: Nothing to be done for `install'.
```

Phony targets should be used as convenient and conventional names. It's helpful for a "help" target to print out useful targets and variables (more on those soon). It's helpful if "all", "install", and "clean" targets are

provided (if relevant), and "all" is the default target, because then many developers will already know how to build and install the software.

Adding prerequisites to a target

An extra `target: prereqs...` line for a target, with no commands associated, will just add prerequisites (which I'll start referring to as "prereqs") to the target. I already used it for adding the `install` prereq to the `.PHONY` target, and it works. I'll add more to the Makefile, to demonstrate:

```
extra.o: extra.h
```

Watch:

```
$ touch extra.h
$ make
gcc -o extra.o -c extra.c
gcc -o program program.o extra.o
building all
$ make
building all
```

This is commonly used with automatically generated dependencies (more on those later).

This only works for prereqs, not for recipes. (Unless you define the rule with `::`, but don't...) The last recipe defined for a target applies (even as all the prereqs from the previous rules are included). When using "pattern rules" (which I'll describe later), which of multiple matching rules applies is more complicated.

Flexibility with variables

Of course, `make` is a bit silly without variables. They're used like this:

```
DESTDIR=
prefix=/usr/local
bindir = $(prefix)/bin
I = install -m 0755

${DESTDIR}${bindir}/program: program
    $I $< $@
```

Variable names longer than a single character need to be quoted with `()` or `{}`, but for single-character variables, the parentheses or braces are optional. Spaces around `=` are optional. I used both quoting styles and both spacing styles in the example, but in real Makefiles I try to be more consistent.

`<` and `@` are special automatic variables inside recipes, referencing the first prerequisite and the target, respectively. I'll go over them later, while introducing "pattern rules".

Variable Precedence

Users can set variables with arguments to `make`. Assuming the above Makefile snippet, you can do:

```
$ make DESTDIR=/tmp/pkg
```

Arguments to `make` will override variables you set in the Makefile. Environment variables can also be used by your Makefile, but they don't override variables set in the Makefile. So

```
$ export DESTDIR=/tmp/pkg
$ make
```

or

```
$ DESTDIR=/tmp/pkg make
```

won't work with the Makefile example above.

One way to take a value from the environment and add to it is by just using a different name:

```
USE_CFLAGS = -g -Wall $(CFLAGS)
program: program.c
    gcc $(USE_CFLAGS) -o program program.c
```

You can also use `?=` instead of `=` to set a variable only if it hasn't already been set (possibly in the environment). In my humble opinion, it's reasonable to force the user to explicitly set variables as arguments, and not generally accept them from the environment (except for really common ones like CFLAGS). I usually avoid `?=`, as well as "override" and other special ways that exist to set variables.

Variable lists and appending

You can append to a variable. Variables can all be thought of as lists of whitespace-separated items, and you can add to the list. Here's an example that exercises some corner cases:

```
USE_CFLAGS = -g -Wall $(CFLAGS)
USE_CFLAGS += -I/usr/local/include -I/opt/tcl-8.5/include

program: program.c
    $(CC) $(USE_CFLAGS) -o program program.c

CFLAGS += -Wextra
```

Notice that you can add to a variable later in the Makefile than where you use it. Variables are expanded as late as possible, so `$(CFLAGS)` in the value of `USE_CFLAGS` isn't expanded until the recipe line using it actually runs, which is after the entire Makefile has been parsed.

The above will also add to CFLAGS from the environment. However, if you specify CFLAGS in an argument to `make`, it will replace the CFLAGS constructed whichever way in the Makefile. Demo:

```
$ ## add -O2 to -Wextra
$ CFLAGS=-O2 make --always-make
cc -g -Wall -O2 -Wextra -I/usr/local/include -I/opt/tcl-8.5/include -o program program.c
$ ## replace -Wextra with -O2
$ make --always-make CFLAGS=-O2
cc -g -Wall -O2 -I/usr/local/include -I/opt/tcl-8.5/include -o program program.c
$ ## no effect
$ USE_CFLAGS=-I/usr/include make --always-make
cc -g -Wall -Wextra -I/usr/local/include -I/opt/tcl-8.5/include -o program program.c
```

(Notice the `--always-make` option I use here, which tells `make` to run recipes for rules even if it considers the targets up-to-date.)

Do not use quotes for variable assignments

Single and double quotes don't work as you're accustomed to from bourne shell scripts - they're treated like any other character. Variables are all lists with spaces separating items. Quotes have no special meaning to `make` (but still have special meaning to the shell when they end up in a recipe line).

Long story short, don't use spaces in file or folder names that `make` works with. Most things really are impossible with spaces in paths. See make meets file names with spaces for a more thorough treatment of this topic.

If there's a space in a parent or grandparent of the source directory, you can avoid the literal name by using relative paths, so in practice this restriction isn't *that* terrible.

Pattern rules

A basic pattern rule will match a part of a file path/name, usually the extension, like this:

```
%.o: %.c
```

```
	$(CC) -o $@ -c $<
```

This will provide a way for `make` to produce a .o file if it can find a corresponding .c file. It can be triggered when something which is being built depends on that .o file, and there isn't a rule with a more specific target to produce that .o file.

Automatic Variables

The following variables can only be used inside the *recipe* commands, but are very convenient and particularly needed for "pattern rules".

- `$@` is the target
- `$^` is all prereqs
- `$<` is the first prereq
- `$*` is the "stem" of the "pattern rule" match

Let's briefly see exactly how that last one works:

```
BLDDIR=build
SRCDIR=src

all: $(BLDDIR)/thing $(BLDDIR)/util/foo

$(BLDDIR)/%: $(SRCDIR)/%.af $(SRCDIR)/%.bf
    @echo weird_cmd --af=$*.af --bf=$*.bf
```

running it:

```
$ mkdir -p src/util
$ touch src/thing.af src/thing.bf src/util/foo.af src/util/foo.bf
$ make
weird_cmd --af=util/foo.af --bf=util/foo.bf
weird_cmd --af=thing.af --bf=thing.bf
```

Using patsubst and other functions

You can transform a list, from one variable, into a related list, with `$(patsubst ...)`. It's very useful, for things like generating a list of object files from a list of source files.

```
SRCS = one.c two.c three.c
SHARED = shared.c util.c
OBJS := $(patsubst %.c, %.o, $(SRCS) $(SHARED))
# $(OBJS) will expand to: one.o two.o three.o shared.o util.o
```

As you can see, `patsubst` takes 3 arguments: a wildcard pattern to apply to elements in a list, an output format for each element based on what the wildcard matched, and the list to process (which can be the result of an expression including variables and other stuff).

There are other functions besides patsubst which you can call this way, for example:

```
PATHVAR = $(dir path/to/file.o) # like dirname
OUTPUTVAR := $(shell mycmd arg1 ...) # run a command in a shell, return stdout
```

I used the := form of variable assignment in some of these cases to ensure that the function or shell fork/exec only happens once, immediately. There could be a performance hit if you use the variable in multiple places, and the variables and functions involved are computed each time. It could cause an "incremental" build (with only some files changed since the last build) to take longer than it should. To be honest, I've never measured it, and in most cases only a shell command which takes a noticeable fraction of a second would really need this.

All together

I usually list just the source files and final build outputs (like binaries), and use the above techniques, plus pattern substitution on lists, to concisely link them together. Here's a realistic (though basic) Makefile:

```
BINARIES = tool_a

# backslash for line continuation, so this list is parsed like a single line
TOOL_A_SRCS = \
tool_a.c      \
util.c        \

prefix = /usr/local
bindir = $(prefix)/bin
DESTDIR =
BLDDIR = build

INSTALL = install
CC = gcc
LD = gcc

TOOL_A_OBJS := $(patsubst %.c, $(BLDDIR)/%.o, $(TOOL_A_SRCS))

# "all" (default target) depends on the built binaries
all: $(patsubst %, $(BLDDIR)/%, $(BINARIES))

# "install" depends on the installed binaries
install: $(patsubst %, $(DESTDIR)$(bindir)/%, $(BINARIES))

clean:
    rm -f $(patsubst %, $(BLDDIR)/%, $(BINARIES))
    rm -f $(TOOL_A_OBJS)
    rmdir $(BLDDIR) || true

# dependencies, additional to the recipes below
$(BLDDIR)/tool_a: $(TOOL_A_OBJS)
$(BLDDIR)/tool_a.o: src/shared.h

$(BLDDIR)/%:
    $(LD) -o $@ $(LDFLAGS) $^

$(BLDDIR)/%.o: src/%.c
    @mkdir -p $(dir $@)
    $(CC) -o $@ $(CFLAGS) -c $<

$(DESTDIR)$(bindir)/%: $(BLDDIR)/%
    @mkdir -p $(dir $@)
    $(INSTALL) $< $@

.PHONY: all install clean
```

A demo of its usage:

```
$ make
gcc -o build/tool_a.o  -c src/tool_a.c
gcc -o build/util.o  -c src/util.c
gcc -o build/tool_a  build/tool_a.o build/util.o
$ touch src/shared.h
$ make
gcc -o build/tool_a.o  -c src/tool_a.c
gcc -o build/tool_a  build/tool_a.o build/util.o
$ make DESTDIR=../pkg install
install build/tool_a ../pkg/usr/local/bin/tool_a
$ make CFLAGS=-O2 BLDDIR=../build-optimized PREFIX=/usr DESTDIR=../pkg install
gcc -o ../build-optimized/tool_a.o -O2 -c src/tool_a.c
gcc -o ../build-optimized/util.o -O2 -c src/util.c
gcc -o ../build-optimized/tool_a  ../build-optimized/tool_a.o ../build-optimized/util.o
install ../build-optimized/tool_a ../pkg/usr/local/bin/tool_a
```

Nifty.

Unexpressed Dependencies

In my example above, there's a type of change that make won't pick up on, causing make to fail to rebuild things that would be different:

- build flags
- the build tool

In the example, I used a different BLDDIR for different CFLAGS. This works, but it's something you have to remember yourself, or put in a wrapper script for your project. You could theoretically serialize the various options, compare them to an "options" file and update it only if needed, and give all build targets an additional prereq - the "options" file. But that would be pretty ugly.

The compiler and linker variables will certainly change the output too. Perhaps you're also cross-compiling for an embedded arm SOC, so you do:

```
$ make CC=arm-unknown-eabi-gcc LD=arm-unknown-eabi-gcc DESTDIR=/home/me/project/image-root install
```

(Easy cross compiles and installs to alternative roots!)

With the example and style of rules I've shown, you'll have to keep separate build directories for different build configurations (like "debug" and "production", or "x86" and "arm"). You'll have to know when you do have to "make clean" after all: when you change the build options or tools used with an existing build output directory. But isn't it nifty that, if you change one C file, you can do incremental builds of all your build configurations, and they all reuse what they can from their separate build directories?

I must admit that some other build tools do keep track of all inputs - variables, source contents, and more. I like `make`'s performance and the relative simplicity of its implementation, and I think these are reasonable tradeoffs in terms of tracking of inputs for a build step.

Making build directories

A simple way to make sure the build (sub-)directory being used exists is to throw in a `@mkdir -p $(dir $@)` at the beginning of each build recipe. That uses a `make` function to get the directory of the target file, and tells mkdir to make that directory and all its parents as necessary, allowing any of them to already exist.

Some implementations of mkdir could report failure, if they don't check for the race condition: if you do a parallel build, and the recipes for multiple targets run mkdir at once, one of them might see that a directory doesn't exist, try to create it, and fail because another mkdir just created it. If your mkdir has this behavior (I wouldn't call it a "problem"), make could fail the build because mkdir reported failure, and that would be annoying. In that case, I might just add `|| true` to the command. The typical linux mkdir (from GNU coreutils) seems to handle this case on its own, and I'd guess that some other unix systems' mkdir implementations do too.

You might think of adding a prerequisite to each target, of the target's directory. Well, directories don't work as targets in `make`. Some people use a proxy file in a directory to represent the directory's existence, for example a `$(DIR)/.keep` file. Now you can add that as a prereq of a target, and the rule for `%/.keep` files can be to just "mkdir" and "touch". I don't prefer this strategy because it creates extra files, and it's awkward for recipes which use the full list of their prerequisites in the commands, particularly recipes for linking binaries or libraries. Also, the cheesy "always mkdir -p" trick doesn't add any overhead to `make` calculating partial rebuilds, and adds little overhead to the actual build steps.

include Makefile parts

You can include aka source parts of Makefiles. I often do this for long lists of sources. To enhance the example above, we could create a `sources.mk`:

```
TOOL_A_SRCS = \
tool_a.c      \
util.c        \
```

and then include it in the Makefile, instead of the list itself:

```
BINARIES = tool_a

include sources.mk
```

(I also tend to wrap those long lists using backslashes, which work similarly in Makefile as in many other languages, including bash. The backslash after the last item doesn't need to be there, but I include it so that adding to the end of the list only changes one line. A final backslash requires a final newline, but you should be putting those in most source files anyway, for the same reason: diffs.)

This plain `include` style causes `make` to abort if the argument filename doesn't exist. You can also write it with a leading hyphen (`-include`), in which case make doesn't complain if the file doesn't exist, it just continues with the rest of the Makefile. In this case, we want the non-hyphen type of include, because sources.mk is now definitely required.

You can also give multiple arguments to `include` or `-include`, all are files to be sourced.

Automatic dependencies for C / C++

Even if you don't use C or C++, this section probably has some tips you would find interesting. If you really want to skip it, jump down to Unintentionally matched pattern rules.

In the example above, I added a line to express a C file dependency on a header:

```
$(BLDDIR)/tool_a.o: src/shared.h
```

You might have many more source files and headers (or equivalent) in your project, and while the rest of the Makefile can handle almost any number of sources in the TOOL_A_SRCS list without any further complication, keeping track of all the header dependencies would be verbose and error prone, if you did it manually. It's pretty easy to let gcc (and probably llvm/clang) do this for you. Here are some changes and additions to the example Makefile above:

```
# sort removes duplicates
DEPS := $(sort $(patsubst %, %.deps, $(TOOL_A_OBJS)))
-include $(DEPS)

# hack to avoid .deps matching the link rule, better fixes explained below
$(DEPS):
    @true

# update clean to also delete *.deps
clean:
    rm -f $(patsubst %, $(BLDDIR)/%, $(BINARIES))
    rm -f $(TOOL_A_OBJS)
    rm -f $(DEPS)
    rmdir $(BLDDIR) || true

$(BLDDIR)/%.o: src/%.c
    @mkdir -p $(dir $@)
    $(CC) -o $@ $(CFLAGS) -c $< -MMD -MF $@.deps
```

I added the flags `-MMD -MF $@.deps` to the compiler command line in the object build step. This causes gcc to emit a Makefile line `target: prereqs ...` into the file `$@.deps`, including all files it read in order to compile the target, but omitting "system" header files. Here's the result:

```
$ cat build/tool_a.o.deps
build/tool_a.o: src/tool_a.c src/shared.h
```

Missing deps files can be OK

I used the non-aborting-on-missing `-include` with multiple deps files arguments. Those files won't exist until after the first build (to a particular build dir, or after `make clean`). For most simple C/C++ projects, that's actually OK. They're only really needed for a rebuild, to know what objects need to be rebuilt if a header changes. For the initial build, all objects will need to be built anyway. This assumes that the header files are not generated during the `make` run, that they're static source files you wrote or copied into your project.

If a header is generated by a recipe of the Makefile, then this isn't good enough: on the first build, `make` could easily try to build the object file before (or concurrently with) running the recipe that generates the required header. To fix that, add a specific dependency just on the generated header, just to make sure it's generated

first, similar to how `shared.h` was handled before adding the automatic deps. If a bunch of files depend on the generated header, you could add something like:

```
$(TOOL_A_OBJS): $(BLDDIR)/config.h
```

(and add `-I$(BLDDIR)` to the CFLAGS.)

System vs non-system headers

If you include "system" header files, by using `-MD` instead of `-MMD`, the output is extensive. Just `<stdio.h>` pulls in 16 glibc and gcc header files on my system. The distinction of "system" vs non-system headers is which ones can be found by the compiler without you specifying the `-I/path/to/dir` flag. Just `/usr/include`, `/usr/local/include` (but not `/usr/local/$LIBNAME/include`), and whatever is in a cross-compiler's sysroot, are "system" include dirs. Everything else, including headers next to the source files or in other directories you specify with `-I`, are not.

In most cases, for me, the library headers in "system" locations are very stable, and I like not having them show up in the deps. That makes the deps files easier to inspect, and probably makes incremental rebuilds faster for large projects. But if you use the system openssl and update the system openssl, you'll have to know to "make clean" for that, or use the more comprehensive deps output of gcc.

Unintentionally matched pattern rules

Finally, there's that odd "null" rule I added:

```
$(DEPS):
    @true
```

in order to fix this:

```
$ make
gcc -o build/util.o.deps
gcc: fatal error: no input files
# build continues, that's not fatal to make
```

`make` implicitly/automatically tries to update any files it `include`s, and reload before making other targets. There happened to exist a pattern rule that matched the .deps files included: the linking (LD) rule. That "null" rule takes precedence, and trivially succeeds. It's a dirty hack though, which I included to show a quick and dirty way to solve a problem, and also to allow the explanation of some better alternatives.

I often "encode" information about a file in the path of the file. So in this case I would differentiate types of targets by their build paths:

```
all: $(patsubst %, $(BLDDIR)/bin/%, $(BINARIES))

$(BLDDIR)/bin/tool_a: $(TOOL_A_OBJS)

$(BLDDIR)/bin/%:
    @mkdir -p $(dir $@)
    $(LD) -o $@ $(LDFLAGS) $^
```

That's sufficient to prevent the pattern rule for linking from matching the .deps files, and the "null" rule is no longer needed.

There's another `make` feature which can fix this problem, and could be more useful in trickier situations. Pattern rules can include the list of targets they're for:

```
BUILT_BINARIES = $(patsubst %, $(BLDDIR)/%, $(BINARIES))
all: $(BUILT_BINARIES)

$(BUILT_BINARIES): $(BLDDIR)/%: $(EMPTY_PREREQS)
    @mkdir -p $(dir $@)
    $(LD) -o $@ $(LDFLAGS) $^
```

If a `make` rule has two `:`, separated by a pattern, then it's a pattern rule that only applies to the targets before the first `:`. I threw in the EMPTY_PREREQS just to show that prereqs could go there, like the end of a normal pattern rule.

Debugging and Built-in Rules

You can use the `-d` flag to `make` to have it print out, in great detail, why it is or is not rebuilding a target. Let's try it out:

```
$ touch src/shared.h
$ make -d
GNU Make 4.0
... # yada yada
Reading makefiles...
Reading makefile 'Makefile'...
Reading makefile 'sources.mk' (search path) (no ~ expansion)...
Reading makefile 'build/tool_a.o.deps' (search path) (don't care) (no ~ expansion)...
Reading makefile 'build/util.o.deps' (search path) (don't care) (no ~ expansion)...
Updating makefiles....
 Considering target file 'build/util.o.deps'.
  Looking for an implicit rule for 'build/util.o.deps'.
  Trying pattern rule with stem 'util.o.deps'.
  Trying implicit prerequisite 'build/util.o.deps.o'.
  Trying pattern rule with stem 'util.o.deps'.
  Trying implicit prerequisite 'build/util.o.deps.c'.
  Trying pattern rule with stem 'util.o.deps'.
... # yada yada
  Trying implicit prerequisite 'build/util.o.deps,v'.
  Trying pattern rule with stem 'util.o.deps'.
  Trying pattern rule with stem 'util.o.deps'.
  Trying implicit prerequisite 'build/RCS/util.o.deps'.
  Trying pattern rule with stem 'util.o.deps'.
  Trying implicit prerequisite 'build/s.util.o.deps'.
  Trying pattern rule with stem 'util.o.deps'.
  Trying implicit prerequisite 'build/SCCS/s.util.o.deps'.
  Trying pattern rule with stem 'util.o.deps'.
  Trying implicit prerequisite 'build/util.o.deps.o'.
  Looking for a rule with intermediate file 'build/util.o.deps.o'.
   Avoiding implicit rule recursion.
   Trying pattern rule with stem 'util.o.deps'.
   Trying implicit prerequisite 'src/util.o.deps.c'.
   Trying pattern rule with stem 'util.o.deps'.
   Trying implicit prerequisite 'build/util.o.deps.c'.
... # this goes on for THOUSANDS of lines
```

See, `make` comes with many built-in rules (and built-in variables), and it checks if any of those built-in rules might be able to update any prerequisite, even if those prerequisites already exist and you have written no rules that create them. It looks for source files available in two different ancient source control systems, SCCS and RCS (after which came CVS, SVN, ...), it checks if maybe you have a parser generator description it could turn into C source with "bison", and many more things. The built-in rules which will never match anything annoy me, and the ones which might match something kind of scare me. The rules you write in your Makefile take precedence, but I dislike unintentional things happening during my builds between coincidentally similarly named files. Luckily, `make` comes with an option to turn off the built-in rules: `-r`.

```
$ touch src/shared.h
$ make -r -d
GNU Make 4.0
... # yada yada
Reading makefile 'Makefile'...
Reading makefile 'sources.mk' (search path) (no ~ expansion)...
Reading makefile 'build/tool_a.o.deps' (search path) (don't care) (no ~ expansion)...
Reading makefile 'build/util.o.deps' (search path) (don't care) (no ~ expansion)...
Updating makefiles....
 Considering target file 'build/util.o.deps'.
  Looking for an implicit rule for 'build/util.o.deps'.
  No implicit rule found for 'build/util.o.deps'.
  Finished prerequisites of target file 'build/util.o.deps'.
 No need to remake target 'build/util.o.deps'.
 Considering target file 'build/tool_a.o.deps'.
... # yada yada
```

```
Considering target file 'all'.
 File 'all' does not exist.
  Considering target file 'build/bin/tool_a'.
   Looking for an implicit rule for 'build/bin/tool_a'.
   Trying pattern rule with stem 'tool_a'.
   Found an implicit rule for 'build/bin/tool_a'.
    Considering target file 'build/tool_a.o'.
     Looking for an implicit rule for 'build/tool_a.o'.
     Trying pattern rule with stem 'tool_a'.
     Trying implicit prerequisite 'src/tool_a.c'.
     Found an implicit rule for 'build/tool_a.o'.
      Considering target file 'src/tool_a.c'.
       Looking for an implicit rule for 'src/tool_a.c'.
       No implicit rule found for 'src/tool_a.c'.
       Finished prerequisites of target file 'src/tool_a.c'.
      No need to remake target 'src/tool_a.c'.
      Pruning file 'src/tool_a.c'.
... # yada yada
     Prerequisite 'src/shared.h' is newer than target 'build/tool_a.o'.
    Must remake target 'build/tool_a.o'.
Putting child 0x20b4560 (build/tool_a.o) PID 14793 on the chain.
Live child 0x20b4560 (build/tool_a.o) PID 14793
Reaping winning child 0x20b4560 PID 14793
gcc -o build/tool_a.o  -c src/tool_a.c -MMD -MF build/tool_a.o.deps
Live child 0x20b4560 (build/tool_a.o) PID 14794
Reaping winning child 0x20b4560 PID 14794
Removing child 0x20b4560 PID 14794 from chain.
    Successfully remade target file 'build/tool_a.o'
... # this goes on for less than 60 more lines
```

Wow is that better. Built-in variables can be disabled with `-R`, but they're not nearly as much of a nusiance. If you have a build script that's calling make for you, have it use the `-r` flag. You can also throw `MAKEFLAGS += r` at the top of your Makefile, and it'll work (it's a somewhat magical variable).

On the plus side, you can use `make` to build a stand-alone C file, without any Makefile at all! Try this in an empty folder:

```
$ echo 'int main() { return 42; }' >program_name.c
$ make program_name
cc     program_name.c   -o program_name
```

You can also see the list of built-in rules and variables (and any rules or variables in the local Makefile, and any environment variables...) with `make -p`. It's rather verbose.

Finally, you can use the `warning` function in `make` as a built-in printf for debugging:

```
$ head Makefile
MAKEFLAGS += r
$(warning MAKEFLAGS: $(MAKEFLAGS))
...
$ make -k
Makefile:2: MAKEFLAGS: k r
gcc -o build/util.o  -c src/util.c -MMD -MF build/util.o.deps
...
```

The GNU make Manual is a good reference for all this stuff and more, but also use little test Makefiles with warnings or debug flags to see how a `make` feature really works.

Conventions

Try to tastefully pick some of the Makefile Conventions from the GNU Make manual, mostly the names of the most common targets and variables, like "DESTDIR", "prefix", "all", and "install".

configure scripts

You've probably run a `./configure` script before, just before running `make` to build a package. Most of them are created by autoconf, a part of a suite of tools often referred to as "autotools", also written by the GNU project. I'd suggest not trying to figure that all out, not right away (maybe not ever). But you can create your own

configure script which emulates the most commonly used features of the configure scripts created by autoconf. A configure script can ideally be run *from* a separate build folder, like so:

```
$ mkdir ../thing-build && cd ../thing-build
$ ../thing-1.0/configure --prefix=/usr/local
$ make
```

The configure script would create a local Makefile, and embed variables for the relative source path, install prefix, and anything else specified to configure. A minimal implementation might then just `include` the real body of the Makefile from "../thing-1.0/Makefile". Or something like that - a post longer than this one could probably be written about detecting platform features and generating Makefiles, and it might involve an even older and more esoteric language called "M4". But in many cases, a plain Makefile can do everything you want, without too much implementation complexity.

Conditional sections

As an aside about multiple platform support, I'll show one more feature of `make` which I've used for that purpose:

```
SO_NAME = so
SO_CMD = -soname
SO_EXT_MAJOR = $(SO_NAME).$(SO_VER_MAJOR)
SO_EXT = $(SO_NAME).$(SO_VER)
UNAME := $(shell uname -s)
ifeq ($(UNAME),Darwin)
SO_NAME = dylib
SO_CMD = -install_name
SO_EXT_MAJOR = $(SO_VER_MAJOR).$(SO_NAME)
SO_EXT = $(SO_VER).$(SO_NAME)
endif
```

That's from the Makefile for dablooms, to enable it to build a shared library on OS X in addition to linux. Notice the `ifeq (x,y)` and `endif` - that's one of the better ways to conditionally set variables from inside the Makefile, instead of from the user or script which invokes `make`.

Recursive Make

Finally, it's somewhat common for a make rule to run make again in a subdirectory, like so:

```
docs:
    $(MAKE) -C $(SRCDIR)/docs
```

I'd suggest not doing that. There's a well-known paper about the practice: Recursive Make Considered Harmful. The take-away is that doing this can result in incomplete dependency graphs being used, and makes builds slower. The paper is rather old now, and things have changed a bit, but I should give it credit for introducing me to a better understanding of `make`.

Thanks for reading

This blog post turned out a lot longer than I expected when I started out. The principles of `make` seem simple when you're familiar with them, but even if they are, the details you run into can be numerous. Despite that (and despite using C for most of the examples), I hope this post has shown how `make` can be very useful, and not really *that* bad.

I'd like to thank Dave Marchevsky and Oliver Hardt for reviewing this. (Go ahead and blame them for any grammatical errors that slipped by, but the silly content is mine.)

- by pierce

Powered by Pelican. Theme based on "notmyidea" by Smashing Magazine, thanks!