Build Tools, Makefiles, and Dependencies

Audience for Build Tools

- **Coders:** These people write the actual source code.
- **Builders:** These people are in charge of assembling the pieces of the program into a coherent application. They "run make".
- **Distributors:** These people take the built products and ship them out to the users. They "run Debian".
- **Installers:** These people take the output of the distributors and install them on the machines.
- **Testers:** These people try to find problems in the code or any of the steps after it like distribution and installation. These are often done by people lower in the chain, like interns.
- **Users:** These people are the intended consumers of the product.

No Build Tools?

You write your own project-specific build tools. Suppose you have a simple example of multiple C files that you need to link together. You could write a build script `build.sh`:

```
gcc -c x.c
gcc -c y.c
gcc -c z.c
gcc x.o y.o z.o -o foo
```

> Your goal as a software developer is to automate yourself out of a job. You constantly want to take the things that you're doing and automate it. **- Dr. Eggert, probably**

Make and Makefiless

A simple build tool intended to be a level up from a shell script is **Make**. This was motivated from scripts taking too long to execute. They were inefficient because they would compile the entire program even if only one file was edited, so it does a lot of unnecessary work.

Thus, the essence of the **Makefile** is that you write a set of **recipes** in it, each of which instructing how to build a file. When you run `make`, the program does the *minimum amount of work* necessary to build the file. It does this with the concept of **prerequisites**.

Anatomy of a Makefile

Make is like a hybrid language. The `target: prerequisites` lines are its own "Makefile" language. The recipes are in the shell language, commands that you would write at the terminal:

```
CC = gcc
x.o: x.c
  $(CC) -c x.c
y.o: y.c
  $(CC) -c y.c
```

```
z.o: z.c
  $(CC) -c z.c
foo: x.o y.o z.o
  $(CC) $< -o $@
  # Escaping the $ so the shell can see it
  echo $$PATH
```

Makefiles support special variables within a rule: `$@` is the target of the rule and `$^` is the prerequisites of the rule. For example:

```
foo: foo1.o foo2.o foo3.o
  g++ $^ -o $@
```

Adding a `-` before a command means to keep going even if that command fails. This can be useful in cleaning commands, like:

```
distclean: distclean-recursive
  -rm -f $(am_CONFIG_DISTCLEANFILES)
  -rm -f Makefile
```

Macros

```
XYZ = foo1.o foo2.o foo3.o  # XYZ is a macro
foo: XYZ
  g++ XYZ -o foo
```

Commonly you'll see people define `CC` for the compiler command (`gcc`) and `CFLAGS` for the compiler options:

```
CC = gcc
CFLAGS = -O2 -g3 -Wall -Wextra
```

Phony Targets

A **phony target** is a rule that has NO output file. The most common example is the `clean` pattern:

```
clean:
  rm *.o
```

Then use `.PHONY` to declare a phony target:

```
.PHONY: clean
```

Running Make

Make is smart enough to see the dependencies (prerequisites) and avoid doing work that has already been done. When you run:

```
make foo
```

It looks at the timestamp of `foo` and the timestamps of its prerequisites, `x.o`, `y.o`, `z.o`. to determine what files have changed since last run (last modified time of each file). If a file is up-to-date, it doesn't need to touch it.

You can run a specific rule to run by specifying the target at the command line:

```
make z.o
```

You can override macros at the command line with similar `=` syntax:

```
make CC=clang
```

Common Mistakes

Often times, bugs arise from specifying the *dependencies* of the targets.

- A *missing* dependency means the product may be wrong. Make will falsely assume its job is done if it does not know if needs to act on a certain file.
- An *extraneous* dependency is a more benign mistake. Files are still compiled the same way as before, but it may cause Make to do extra work.

Parallel Makes

Suppose you have a big project.

```
project/
  Makefile
  sub1/
    Makefile
  sub2/
    Makefile
  sub3/
    Makefile
```

One approach is to have Makefiles in each subdirectory. In this case, the root Makefile *delegate* work to the subsidiary Makefiles:

```
test:
  # not sure if this is right syntax
  for i in a b c; do; (cd $$ && make); done
```

- ➕ This modularizes the build process.
- ➖ However, the parent Makefile is **sequential** code, and you don't have the opportunity to run multiple rules in parallel.

Another approach is to have one big Makefile at the project root will all the rules, including tests for anything under subdirectories. Nowadays, this is the more common pattern:

- ➕ It is a performance bonus, and it is usually automatically generated anyway.
- ➖ Extraneous/incorrect dependencies may slow down the process because there's less opportunity for parallelism. A missing dependency is even worse because they can cause different builds affecting different files, resulting in crashes.

Building Makefiles

Suppose you have code that you want to tailor to a certain implementation. In this particular example, the C header `<stdckdint.h>` is not on SEASnet yet, but suppose you want to be able to compile code that uses it anyway.

A common convention is to have a shell script `./configure`, which *generates* Makefiles from a template, similar to the concept of a *compiler*.

```
# This syntax here is called a heredoc
cat > checkfile.c << EOF
<-- code that uses the feature -->
EOF

if gcc checkfile.c; then
  echo "#define HAVE_STDCKDINT_H 1" >> config.h
fi
```

Then in your source code:

```
#include <config.h>
int main(void)
{
  #if HAVE_STDCKDINT_H
    use the features for stdckdint.h
  #endif
}
```

Then you define the `builder` script to first *make* the Makefile and cause other configuration side effects, and then run the Makefile:

```
./configure
make
```

Autoconf and Automake

Autoconf is a program that generates `./configure` files. Just as how Makefiles are automatically generated by `./configure`, the `./configure` file is generated by Autoconf. We see that build tools often build on top of each other, each generated by the next. This is a common pattern in software construction.

Automake is like a front-end abstraction for Autoconf, so like yet another level up from Autoconf.

Dependencies

You can think of them in terms of **dependency graphs**, which are typically (and ideally) *acyclic*.

Two ways dependencies come up in software construction:

Build-time Dependency

"File x depends on file y." This can be expressed simply in Makefile syntax:

```
x: y
    # create x; y is input
```

As a project scales, the dependency lists can get much longer.

```
foo.o: foo.c x.h y.h z.h
    gcc -c foo.c $^
```

Maybe you extend the list based on what you see from the `#include` lines in the source files.

This can be problematic because you need to make sure that the dependencies in the Makefile *match* those of the source code, else it would slow down your building at best and break your it if a dependency is missing.

Thus, instead of maintaining dependencies *by hand*, you should *calculate* them with some `make` preprocessor - a tool to automatically generate your Makefiles for you.

Packaging Dependency

Each package can be installed independently, BUT:

- Packages can depend on other packages

- Pnd every package can be version-specific.

This can get complicated, so ideally you'd want a **package manager**.

You can view the dependencies of a program, using `grep` as an example:

```
$ dnf deplist grep
Not root, Subscription Management repositories not updated
Last metadata expiration check: 1:16:41 ago on Mon 05 Dec 2022 04:57:02 PM PST.
package: grep-3.1-6.el8.x86_64
  dependency: /bin/sh
   provider: bash-4.4.20-4.el8_6.x86_64
  dependency: /sbin/install-info
   provider: info-6.5-7.el8.x86_64
  dependency: libc.so.6()(64bit)
   provider: glibc-2.28-211.el8.x86_64
  dependency: libc.so.6(GLIBC_2.14)(64bit)
   provider: glibc-2.28-211.el8.x86_64
  dependency: libc.so.6(GLIBC_2.2.5)(64bit)
   provider: glibc-2.28-211.el8.x86_64
  dependency: libc.so.6(GLIBC_2.3)(64bit)
   provider: glibc-2.28-211.el8.x86_64
  dependency: libc.so.6(GLIBC_2.3.4)(64bit)
   provider: glibc-2.28-211.el8.x86_64
  dependency: libc.so.6(GLIBC_2.4)(64bit)
   provider: glibc-2.28-211.el8.x86_64
  dependency: libc.so.6(GLIBC_2.5)(64bit)
   provider: glibc-2.28-211.el8.x86_64
  dependency: libpcre.so.1()(64bit)
   provider: pcre-8.42-6.el8.x86_64
  dependency: rtld(GNU_HASH)
   provider: glibc-2.28-211.el8.i686
   provider: glibc-2.28-211.el8.x86_64
```

And the dynamically linked libraries:

```
$ ldd $(which grep)
linux-vdso.so.1 (0x00007fff5ab88000)
        libpcre2-8.so.0 => /lib64/libpcre2-8.so.0 (0x00007f365a1fd000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f3659e38000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f3659c18000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f365a481000)
```

We notice that for example, `grep` requires `sh` because `egrep` utilizes a *shell script*. `grep` also needs the C library.