Implementing non-recursive make

1 Introduction

If you're a regular Unix Make user and you haven't read the excellent paper Recursive Make Considered Harmful by Peter Miller yet, consider doing so first, and then return here. I'm not going to present the issues in that paper here again, but I will to show how you can put the ideas described there to use, without some of the drawbacks that come with the implementation given by the paper.

2 Practical problems

The most basic problem to overcome when implementing single-session make is to avoid flattening your directory structure, while joining the dependency information present in each subdirectory in a single tree. That is, you not only want to keep your subdirectories around, but also want to keep them separate as far as their contents are concerned. You should be able to have both an ant/localfile.h and a bee/localfile.h, and a way for the sources in ant and bee to include the local header, without knowing which directory it's in.

That means, among other things, that you'll want to avoid having to put every directory in your tree in your include path. And simply using local includes, #include "localfile.h", does not solve the problem either, at least not portably. That is because while building, the current directory as seen by the compiler remains at the root of your project tree, and a lot of C preprocessors will look there and not in the directory the current source file is in when they see a local include.

The same story can be held for the Makefile fragments you put in each directory. Putting absolute paths in them is not an option, because you loose the freedom to move subtrees around.

The original paper doesn't address those issues, but using GNU Make, I believe they can be handled effectively.

3 The solution

Basically, what's needed to tackle these issues is a variable that tracks the 'current directory' while the source tree is traversed and makefile fragments are included. This variable can then be used in describing dependency relations in a relative fashion, and in the include path for the compiler in build recipes.

Two problems arise with that. The first one being that you need a stack-like construct to save the variable that holds the current directory while traversing the source tree using makefile includes.

The second is that include paths must be specified on the compiler command line, that is in the build recipes. However, variables used in recipes are only substituted at build time, so the various values that the current directory-variable had while assembling the dependency tree, will be long forgotten. In other words, how do we get the "current" directory on the compiler command line?

Luckily, GNU Make provides two features help solve those problems. Using target-specific variable assignments (only correctly implemented in version 3.79 and later), we can get the "current" directory into the build recipes. And using string concatenation, suffix stripping and double indirection, we can implement a stack to save the current directory on while including Makefile fragments in subdirectories.

4 How to put that together

These examples are all taken from OpenRADIUS 0.9.5, which uses the concepts described here in its build system.

4.1 The main Makefile

Basically, we start out with a regular Makefile, which defines a few things, such as the tools we'll use, like this:

```
### Build flags for all targets
#
CF_ALL          = -g -Wall
LF_ALL          =
LL_ALL          =

### Build tools
#
CC              = ./build/ccd-gcc
```

```
INST        = ./build/install
COMP            = $(CC) $(CF_ALL) $(CF_TGT) -o $@ -c $<
LINK            = $(CC) $(LF_ALL) $(LF_TGT) -o $@ $^ $(LL_TGT) $(LL_ALL)
COMPLINK        = $(CC) $(CF_ALL) $(CF_TGT) $(LF_ALL) $(LF_TGT) -o $@ $< $(LL_TGT) $(LL_ALL)

### Standard parts
#
include Rules.mk
```

Fig. 1: The main Makefile

After these macros, which can be platform- and configuration specific, you continue by including the generic Rules.mk, which will include additional Rules.mk files in each subdirectory. First we examine the tools used above a little closer though.

4.2 The compile script

The command used as `$(CC)` above is a wrapper script that invokes the compiler and outputs any extra dependency information that is found while compiling, in a format suitable for inclusion in Makefile fragments. Some compilers can do that on the fly, but omit directory names or put things in the wrong place in other ways; other systems will need a separate invocation of the preprocessor or even a completely separate tool.

See build/ccd-gcc for a version of the script tailored for GCC, and build/ccd-osf for a version that uses Compaq's C compiler. The idea in each case being that the script should hide the details of automatic dependency generation from the rest of the build system.

If you want to build a script for your own compiler, the requirement is that it allows both compiling and compiling and linking in one step, and that a .d file must be generated containing the target's (extra) dependencies, with the same base name as the target which is specified using the -o command line option.

For example, assume dir/file1.c includes header1.h and header2.h locally. Then,

```
ccd-mycc -o dir/some_executable dir/file1.c dir/file2.c otherdir/file3.o otherdir/file4.a
```

must generate both the executable and a file dir/some_executable.d, which contains all the dependencies for the target, and each prerequisite as an empty target on its own, so that GNU Make will silently consider the target out of date if a prerequisite doesn't exist:

```
dir/some_executable dir/some_executable.d: dir/file1.c dir/file2.c \
    otherdir/file3.o otherdir/file4.a dir/header1.h dir/header2.h
dir/file1.c:
dir/file2.c:
otherdir/file3.o:
otherdir/file4.a:
dir/header1.h:
dir/header2.h:
```

Of course, the dependencies outside of the headers are entirely optional, because in they will already be given by the makefile fragment, but most compilers or mkdep tools will generate them anyway, and including them doesn't do any harm. No need to take them out.

4.3 The install script

A very irritating aspect of trying to build portable packages is the fact that the Unix "install" command is highly non-portable, and that the few portable features hardly surpass those of "cp". Finding GNU Automake's install-sh not very satisfactory either, I decided to write a very powerful yet completely portable install command as a POSIX /bin/sh script. It creates directories recursively where needed, but does not assume a working mkdir -p command. It's simple but versatile.

See the documentation included at the top of the build/install script itself.

4.4 The top-level Rules.mk

As can be seen at the bottom of the main makefile, a file named Rules.mk is included, which starts the recursive include process. It also contains a few basic recipes and specifies the final targets:

```
# Standard things
```

```
.SUFFIXES:
.SUFFIXES:  .c .o

all:        targets


# Subdirectories, in random order

dir := ant
include     $(dir)/Rules.mk
dir := bee
include     $(dir)/Rules.mk


# General directory independent rules

%.o:        %.c
        $(COMP)

%:      %.o
        $(LINK)

%:      %.c
        $(COMPLINK)


# The variables TGT_*, CLEAN and CMD_INST* may be added to by the Makefile
# fragments in the various subdirectories.

.PHONY:     targets
targets:    $(TGT_BIN) $(TGT_SBIN) $(TGT_ETC) $(TGT_LIB)

.PHONY:     clean
clean:
        rm -f $(CLEAN)

.PHONY:     install
install:    targets
        $(INST) $(TGT_BIN) -m 755 -d $(DIR_BIN)
        $(CMD_INSTBIN)
        $(INST) $(TGT_SBIN) -m 750 -d $(DIR_SBIN)
        $(CMD_INSTSBIN)
ifeq ($(wildcard $(DIR_ETC)/*),)
        $(INST) $(TGT_ETC) -m 644 -d $(DIR_ETC)
        $(CMD_INSTETC)
else
        @echo Configuration directory $(DIR_ETC) already present -- skipping
endif
        $(INST) $(TGT_LIB) -m 750 -d $(DIR_LIB)
        $(CMD_INSTLIB)


# Prevent make from removing any build targets, including intermediate ones

.SECONDARY: $(CLEAN)
```

Fig. 2: A top-level Rules.mk

4.5 The per-directory makefile fragments

The real work is done by the Rules.mk files present in the directories that hold actual source files. As an example, let's examine the Rules.mk that is in OpenRADIUS 0.9.5's subdirectory common. That directory builds an archive (.a) or library of frequently reused objects. Each Rules.mk begins with a standard part that saves the variable `$(d)` that holds the current directory on the stack, and sets it to the current directory given in `$(dir)`, which was passed as a parameter by the parent Rules.mk:

```
# Standard things
sp      := $(sp).x
dirstack_$(sp)  := $(d)
d       := $(dir)

(Then, if this directory has any subdirectories of its own, we list them here:)

# Subdirectories, in random order

dir := $(d)/test
include     $(dir)/Rules.mk
```

(Next, we create an immediately evaluated variable $(OBJS_common) that holds all the objects in that directory. We also create $(DEPS_common), which lists any automatic dependency files generated later on. To the global variable $(CLEAN), we add the files that the rules present here may create, i.e. the ones we want deleted by a make clean command.)

```
# Local variables

OBJS_$(d)   := $(d)/debug.o $(d)/md5.o $(d)/misc.o \
           $(d)/ringbuf.o $(d)/textfile.o $(d)/subprocs.o \
           $(d)/metadata.o $(d)/metatype.o \
           $(d)/metadict.o $(d)/metaops.o
DEPS_$(d)   := $(OBJS_$(d):%=%.d)

CLEAN       := $(CLEAN) $(OBJS_$(d)) $(DEPS_$(d)) \
           $(d)/common.a $(d)/platform.h
```

(Now we list the dependency relations relevant to the files in this subdirectory. Most importantly, while generating the objects listed here, we want to set the target-specific compiler flags `$(CF_TGT)` to include a flag that adds this directory to the include path, so that local headers may be included using `#include <localfile.h>`, which is, as said, more portable than local includes when the source directory is not the current directory.)

```
# Local rules

$(OBJS_$(d)):   CF_TGT := -I$(d)

$(d)/common.a:  $(OBJS_$(d))
        $(ARCH)
```

(Here we try to include any automatically generated dependency files for the objects we know of, but we don't worry if they haven't been generated yet. In that case the .o files shouldn't exist either, so rebuilding will happen anyway, which also generates the .d files.

As a last step, we restore the value of the current directory variable `$(d)` by taking it from the current location on the stack, and we "decrement" the stack pointer by stripping away the `.x` suffix we added at the top of this file.)

```
# Standard things

-include    $(DEPS_$(d))

d       := $(dirstack_$(sp))
sp      := $(basename $(sp))
```

Fig. 3: An example of a Rules.mk file

After doing all this, we can have other files safely depend on `common/common.a`; we know that all dependency relations below it are fully described. As a last example, let's examine another Rules.mk, which is used to generate OpenRADIUS' radclient tool:

```
# Standard things

sp      := $(sp).x
dirstack_$(sp)  := $(d)
d       := $(dir)

# Local rules and targets

TGTS_$(d)   := $(d)/radclient
```

```
DEPS_$(d)   := $(TGTS_$(d):%=%.d)

TGT_BIN     := $(TGT_BIN) $(TGTS_$(d))
CLEAN       := $(CLEAN) $(TGTS_$(d)) $(DEPS_$(d))

$(TGTS_$(d)):   $(d)/Rules.mk

$(TGTS_$(d)):   CF_TGT := -Icommon -DRADDB=\"$(DIR_ETC)\"
$(TGTS_$(d)):   LL_TGT := $(S_LL_INET) common/common.a
$(TGTS_$(d)):   $(d)/radclient.c common/common.a
        $(COMPLINK)

# Standard things

-include    $(DEPS_$(d))

d       := $(dirstack_$(sp))
sp      := $(basename $(sp))
```

Fig. 4: A Rules.mk file for an executable target

As you can see, referring to files in directories other than the local one still needs to be done using a paths relative to the root of the project tree. I don't see that as much of a problem; a little compartimentalization is in most cases exactly what you want to want to achieve by splitting your project across subdirectories. If you really want to refer to a directory relative to the local one, you could use a construct like `$(d)/../common/common.a`.

The most important thing here though, is that you can see that this Makefile fragment and the source files mentioned in it can remain completely unaware of their whereabouts in the project tree. And that's an important maintainability feature, which was left unaddressed by the solutions in "Recursive Make Considered Harmful".