Recursive Make Considered Harmful

Peter Miller
`millerp@canb.auug.org.au`

ABSTRACT

For large UNIX projects, the traditional method of building the project is to use recursive *make.* On some projects, this results in build times which are unacceptably large, when all you want to do is change one file. In examining the source of the overly long build times, it became evident that a number of apparently unrelated problems combine to produce the delay, but on analysis all have the same root cause.

This paper explores an number of problems regarding the use of recursive *make,* and shows that they are all symptoms of the same problem. Symptoms that the UNIX community have long accepted as a fact of life, but which need not be endured any longer. These problems include recursive *make*s which take ''forever'' to work out that they need to do nothing, recursive *make*s which do too much, or too little, recursive *make*s which are overly sensitive to changes in the source code and require constant `Makefile` intervention to keep them working.

The resolution of these problems can be found by looking at what *make* does, from first principles, and then analyzing the effects of introducing recursive *make* to this activity. The analysis shows that the problem stems from the artificial partitioning of the build into separate subsets. This, in turn, leads to the symptoms described. To avoid the symptoms, it is only necessary to avoid the separation; to use a single `Makefile` for the whole project.

This conclusion runs counter to much accumulated folk wisdom in building large projects on UNIX. Some of the main objections raised by this folk wisdom are examined and shown to be unfounded. The results of actual use are far more encouraging, with routine development performance improvements significantly faster than intuition may indicate. The use of a single project `Makefile` is not as difficult to put into practice as it may first appear.

1. Introduction

For large UNIX software development projects, the traditional methods of building the project use what has come to be known as ''recursive *make*.'' This refers to the use use a hierarchy of directories containing source files for the modules which make up the project, where each of the sub-directories contains a *Makefile* which describes the rules and instructions for the *make* program. The complete project build is done by arranging for the top-level `Makefile` to change directory into each of the sub-directories and recursively invoke *make.*

This paper explores some significant problems encountered when developing software projects using the recursive *make* technique. A simple solution is offered, and some of the implications of that solution are explored.

Recursive *make* results in a directory tree which looks something like this:

```
Project
├─ Makefile
├─ module1
│   ├─ Makefile
│   ├─ source1.c
│   └─ etc...
└─ module2
    ├─ Makefile
    ├─ source2.c
    └─ etc...
```

This hierarchy of modules can be nested arbitrarily deep. Real-world projects often use two- and three-level structures.

1.1. Assumed Knowledge

This paper assumes that the reader is familiar with developing software on UNIX, with the *make* program, and with the issues of C programming and include file dependencies.

This paper assumes that you have installed GNU Make on your system and are moderately familiar with its features. Some features of *make* described below may not be available if you are using the limited version supplied by your vendor.

2. The Problem

There are numerous problems with recursive *make*, and they are usually observed daily in practice. Some of these problems include:

- It is very hard to get the *order* of the recursion into the sub-directories correct. This order is very unstable and frequently needs to be manually "tweaked." Increasing the number of directories, or increasing the depth in the directory tree, cause this order to be increasingly unstable.
- It is often necessary to do more than one pass over the sub-directories to build the whole system. This, naturally, leads to extended build times.
- Because the builds take so long, some dependency information is omitted, otherwise development builds take unreasonable lengths of time, and the developers are unproductive. This usually leads to things not being updated when they need to be, requiring frequent "clean" builds from scratch, to ensure everything has actually been built.
- Because inter-directory dependencies are either omitted or too hard to express, the `Makefiles` are often written to build *too much* to ensure that nothing is left out.
- The inaccuracy of the dependencies, or the simple lack of dependencies, can result in a product which is incapable of building cleanly, requiring the build process to be carefully watched by a human.

3. Analysis

Before it is possible to address these seemingly unrelated problems, it is first necessary to understand what *make* does and how it does it. It is then possible to look at the effects recursive *make* has on how *make* behaves.

3.1. Whole Make

Make is an expert system. You give it a set of rules for how to construct things, and a target to be constructed. The rules can be decomposed into pair-wise ordered dependencies between files. *Make* takes the rules and determines how to build the given target. Once it has determined how to construct the target, it proceeds to do so.

Make determines how to build the target by constructing a *directed acyclic graph,* the DAG familiar to many Computer Science students. The vertices of this graph are the files in the system, the edges of this graph are the inter-file dependencies. The edges of the graph are directed because the pair-wise dependencies are ordered; resulting in a *acyclic* graph – things which look like loops are resolved by the direction of the edges.

This paper will use a small example project for its analysis. While the number of files in this example is small, there is sufficient complexity to demonstrate all of the above recursive *make* problems. First, however, the project is presented in a non-recursive form.

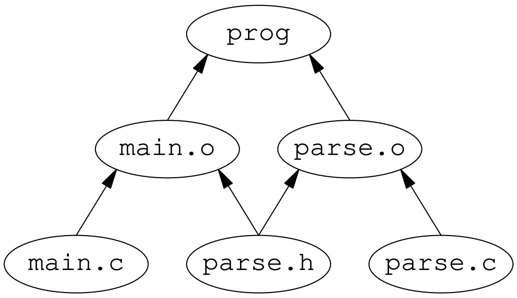
```
Project
├─ Makefile
├─ main.c
├─ parse.c
└─ parse.h
```

The `Makefile` in this small project looks like this:

```
OBJ = main.o parse.o
prog: $(OBJ)
	$(CC) -o $@ $(OBJ)
main.o: main.c parse.h
	$(CC) -c main.c
parse.o: parse.c parse.h
	$(CC) -c parse.c
```

Some of the implicit rules of *make* are presented here explicitly, to assist the reader in converting the `Makefile` into its equivalent DAG.

The above `Makefile` can be drawn as a DAG in the following form:



This is an *acyclic* graph is because of the arrows which express the ordering of the relationship between the files. If there *was* a circular dependency according to the arrows, it would be an error.

Note that the object files (`.o`) are dependent on the include files (`.h`) even though it is the source files (`.c`) which do the including. This is because if an include file changes, it is the object files which are out-of-date, not the source files.

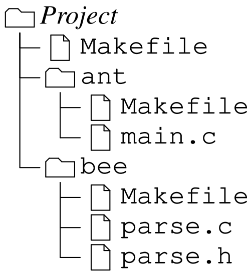
The second part of what *make* does it to perform a *postorder* traversal of the DAG. That is, the dependencies are visited first. The actual order of traversal is undefined, but most *make* implementations work down the graph from left to right for edges below the same vertex, and most projects implicitly rely on this behavior. The last-time-modified of each file is examined, and higher files are determined to be out-of-date if any of the lower files on which they depend are younger. Where a file is determined to be out-of-date, the action associated with the relevant graph edge is performed (in the above example, a compile or a link).

The use of recursive *make* affects both phases of the operation of *make:* it causes *make* to construct an inaccurate DAG, and it forces *make* to traverse the DAG in an inappropriate order.

3.2. Recursive Make

To examine the effects of recursive *make*s, the above example will be artificially segmented into two modules, each with its own `Makefile`, and a top-level `Makefile` used to invoke each of the module `Makefiles`.

The directory structure is as follows:



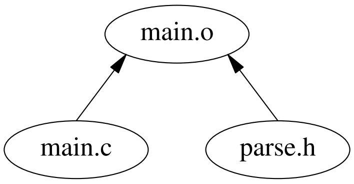
The top-level `Makefile` looks like this:

```
MODULES = ant bee
all:
  for dir in $(MODULES); do \
    (cd $$dir; ${MAKE} all); \
  done
```

The `ant/Makefile` looks like this:

```
all: main.o
main.o: main.c ../bee/parse.h
  $(CC) -I../bee -c main.c
```

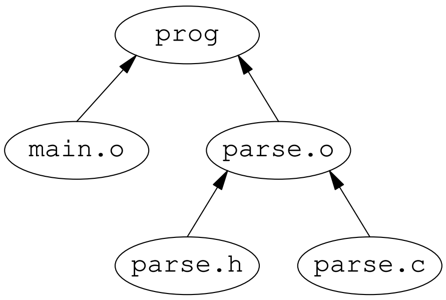
and the equivalent DAG looks like this:



The `bee/Makefile` looks like this:

```
OBJ = ../ant/main.o parse.o
all: prog
prog: (OBJ)
  $(CC) -o $@ $(OBJ)
parse.o: parse.c parse.h
  $(CC) -c parse.c
```

and the equivalent DAG looks like this:

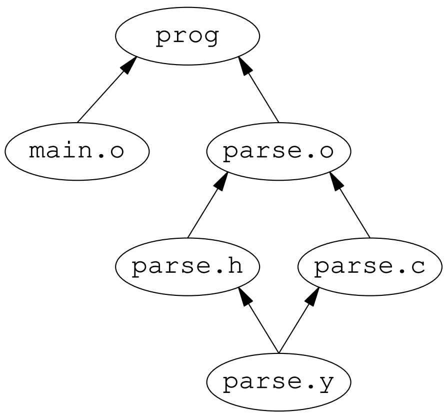


Take a close look at the DAGs. Notice how neither is complete – there are vertices and edges (files and dependencies) missing from both DAGs. When the entire build is done from the top level, everything will work.

But what happens when small changes occur? For example, what would happen of the `parse.c` and `parse.h` files were generated from a `parse.y` yacc grammar? This would add the following lines to the `bee/Makefile`:

```
parse.c parse.h: parse.y
    $(YACC) -d parse.y
    mv y.tab.c parse.c
    mv y.tab.h parse.h
```

And the equivalent DAG changes to look like this:



This change has a simple effect: if `parse.y` is edited, `main.o` will **not** be constructed correctly. The is because the DAG for `ant` knows about only some of the dependencies of `main.o`, and the DAG for `bee` knows none of them.

To understand why this happens, it is necessary to look at the actions *make* will take *from the top level.* Assume that the project is in a self-consistent state. Now edit `parse.y` in such a way that the generated `parse.h` file will have non-trivial differences. However, when the top-level *make* is invoked, first `ant` and then `bee` is visited. But `ant/main.o` is *not* recompiled, because `bee/parse.h` has not yet been regenerated and thus does not yet indicate that `main.o` is out-of-date. It is not until `bee` is visited by the recursive *make* that `parse.c` and `parse.h` are reconstructed, followed by `parse.o`. When the program is linked `main.o` and `parse.o` are non-trivially incompatible. That is, the program is *wrong.*

3.3. Traditional Solutions

There are three traditional fixes for the above "glitch."

3.3.1. Reshuffle

The first is to manually tweak the order of the modules in the top-level `Makefile`. But why is this tweak required at all? Isn't *make* supposed to be an expert system? Is *make* somehow flawed, or did something else go wrong?

To answer this question, it is necessary to look, not at the graphs, but the *order of traversal* of the graphs. In order to operate correctly, *make* needs to perform a *postorder* traversal, but in separating the DAG into two pieces, *make* has not been *allowed* to traverse the graph in the necessary order – instead the project has dictated an order of traversal. An order which, when you consider the original graph, is plain *wrong.* Tweaking the top-level `Makefile` corrects the order to one similar to that which *make* could have used. Until the next dependency is added...

3.3.2. Repetition

The second traditional solution is to make more than one pass in the top-level `Makefile`, something like this:

```
MODULES = ant bee
all:
    for dir in $(MODULES); do \
      (cd $$dir; ${MAKE} all); \
    done
    for dir in $(MODULES); do \
      (cd $$dir; ${MAKE} all); \
    done
```

This doubles then length of time it takes to perform the build. But that is not all: there is no guarantee that two passes are enough! The upper bound of the number of passes is not even proportional to the number of modules, it is instead proportional to the number of graph edges which cross module boundaries.

3.3.3. Overkill

We have already seen an example of how recursive *make* can build too little, but another common problem is to build too much. The third traditional solution to the above glitch is to add even *more* lines to `ant/Makefile`:

```
.PHONY: ../bee/parse.h
../bee/parse.h:
    cd ../bee; \
    make clean; \
    make all
```

This means that whenever `main.o` is made, `parse.h` will always be considered to be out-of-date. All of `bee` will always be rebuilt including `parse.h`, and so `main.o` will always be rebuilt, *even if everything was self consistent.*

4. Prevention

The above analysis is based one one simple action: the DAG was artificially separated into incomplete pieces. This separation resulted in all of the problems familiar to recursive *make* builds.

Did *make* get it wrong? No. This is a case of the ancient GIGO principle: *Garbage In, Garbage Out.* Incomplete `Makefiles` are *wrong* `Makefiles`.

To avoid these problems, don't break the DAG into pieces; instead, use one `Makefile` for the entire project. It is not the recursion itself which is harmful, it is the crippled `Makefiles` which are used in the recursion which are *wrong*. It is not a deficiency of *make* itself that recursive *make* is broken, it does the best it can with the flawed input it is given.

> "*But, but, but... You can't do that!*" I hear you cry. "*A single* `Makefile` *is too big, it's unmaintainable, it's too hard to write the rules, you'll run out of memory, I only want to build my little bit, the build take too long. It's just not practical.*"

These are valid concerns, and they frequently lead *make* users to the conclusion that re-working their build process does not have any short- or long-term benefits. This conclusion is based on ancient, enduring, false assumptions.

4.1. A Single `Makefile` Is Too Big

If the entire project build description were placed into a single `Makefile` this would certainly be true, however modern *make* implementations have *include* statements. By including a relevant fragment from each module, the total size of the `Makefile` and its include files need be no larger than the total size of the `Makefiles` in the recursive case.

4.2. A Single `Makefile` Is Unmaintainable

The complexity of using a single top-level `Makefile` which includes a fragment from each module is no more complex than in the recursive case. Because the DAG is not segmented, this form of `Makefile` becomes less complex, and thus *more* maintainable, simply because fewer "tweaks" are required to keep it working.

Recursive `Makefiles` have a great deal of repetition. Many projects solve this by using include files. By using a single `Makefile` for the project, the need for the "common" include files disappears – the single `Makefile` is the common part.

4.3. It's Too Hard To Write The Rules

The only change required is to include the directory part in filenames in a number of places. This is because the *make* is performed from the top-level directory; the current directory is not the one in which the file appears. Where the output file is explicitly stated in a rule, this is not a problem.

GCC allows a `-o` option in conjunction with the `-c` option, and GNU Make knows this. This results in the implicit compilation rule placing the output in the correct place. Older and dumber C compilers, however, may not allow the `-o` option with the `-c` option, and will leave the object file in the top-level directory (*i.e.* the wrong directory). There are three ways for you to fix this: get GNU Make and GCC, override the built-in rule with one which does the right thing, or complain to your vendor.

Also, K&R C compilers will start the double-quote include path (`#include "`*filename.h*`"`) from the current directory. This will not do what you want. ANSI C compliant C compilers, however, start the double-quote include path from the directory in which the source file appears; thus, no source changes are required. If you don't have an ANSI C compliant C compiler, you should consider installing GCC on your system as soon as possible.

4.4. I Only Want To Build My Little Bit

Most of the time, developers are deep within the project tree and they edit one or two files and then run *make* to compile their changes and try them out. They may do this dozens or hundreds of times a day. Being forced to do a full project build every time would be absurd.

Developers always have the option of giving *make* a specific target. This is always the case, it's just that we usually rely on the default target in the `Makefile` in the current directory to shorten the command line for us. Building ''my little bit'' can still be done with a whole project `Makefile`, simply by using a specific target, and an alias if the command line is too long.

Is doing a full project build every time so absurd? If a change made in a module has repercussions in other modules, because there is a dependency the developer is unaware of (but the `Makefile` is aware of), isn't it better that the developer find out as early as possible? Dependencies like this *will* be found, because the DAG is more complete than in the recursive case.

The developer is rarely a seasoned old salt who knows every one of the million lines of code in the product. More likely the developer is a short-term contractor or a junior. You don't want implications like these to blow up after the changes are integrated with the master source, you want them to blow up on the developer in some nice safe sand-box far away from the master source.

If you want to make ''just your little'' bit because you are concerned that performing a full project build will corrupt the project master source, due to the directory structure used in your project, see the ''Projects *versus* Sand-Boxes'' section below.

4.5. The Build Will Take Too Long

This statement can be made from one of two perspectives. First, that a whole project *make*, even when everything is up-to-date, inevitably takes a long time to perform. Secondly, that these extended delays are unacceptable when a developer wants to quickly compile and link the one file that they have changed.

4.5.1. Project Builds

Consider a hypothetical project with 1000 source (`.c`) files, each of which has its calling interface defined in a corresponding include (`.h`) file with defines, type declarations and function prototypes. These 1000 source files include their own interface definition, plus the interface definitions of any other module they may call. These 1000 source files are compiled into 1000 object files which are then linked into an executable program. This system has some 3000 files which *make* must be told about, and be told about the include dependencies, and also explore the possibility that implicit rules (`.y` → `.c` for example) may be necessary.

In order to build the DAG, *make* must ''stat'' 3000 files, plus an additional 2000 files or so, depending on which implicit rules your *make* knows about and your `Makefile` has left enabled. On the author's humble 66MHz i486 this takes about 10 seconds; on native disk on faster platforms it goes even faster. With NFS over 10MB Ethernet it takes about 10 seconds, no matter what the platform.

This is an astonishing statistic! Imagine being able to a single file compile, out of 1000 source files, in only 10 seconds, plus the time for the compilation itself.

Breaking the set of files up into 100 modules, and running it as a recursive *make* takes about 25 seconds. The repeated process creation for the subordinate *make* invocations take quite a long time.

Hang on a minute! On real-world projects with less than 1000 files, it takes an awful lot longer than 25 seconds for *make* to work out that it has nothing to do. For some projects, doing it in only 25 minutes would be an improvement! The above result tells us that it is not the number of files which is slowing us down (that only takes 10 seconds), and it is not the repeated process creation for the subordinate *make* invocations (that only takes another 15 seconds). So just what *is* taking so long?

The traditional solutions to the problems introduced by recursive *make* often increase the number of subordinate *make* invocations beyond the minimum described here; *e.g.* to perform multiple repetitions (3.3.2), or to overkill cross-module dependencies (3.3.3). These can take a long time, particularly when combined, but do not account for some of the more spectacular build times; what else is taking so long?

Complexity of the `Makefile` is what is taking so long. This is covered, below, in the *Efficient Makefiles* section.

4.5.2. Development Builds

If, as in the 1000 file example, it only takes 10 seconds to figure out which one of the files needs to be recompiled, there is no serious threat to the productivity of developers if they do a whole-project *make* as opposed to a module-specific *make*. The advantage for the project is that the module-centric developer is reminded at relevant times (and only relevant times) that their work has wider ramifications.

By consistently using C include files which contain accurate interface definitions (including function prototypes), this will produce compilation errors in many of the cases which would result in a defective product. By doing whole-project builds, developers discover such errors very early in the development process, and can fix the problems when they are least expensive.

4.6. You'll Run Out Of Memory

This is the most interesting response. Once long ago, on a CPU far, far away, it may even have been true. When Feldman [1] first wrote *make* it was 1978 and he was using a PDP11. Unix processes were limited to 64KB of data.

On such a computer, the above project with its 3000 files detailed in the whole-project `Makefile`, would probably *not* allow the DAG and rule actions to fit in memory.

But we are not using PDP11s any more. The physical memory of modern computers exceeds 10MB for *small* computers, and virtual memory often exceeds 100MB. It is going to take a project with hundreds of thousands of source files to exhaust virtual memory on a *small* modern computer. As the 1000 source file example takes less than 100KB of memory (try it, I did) it is unlikely that any project manageable in a single directory tree on a single disk will exhaust your computer's memory.

4.7. Why Not Fix The DAG In The Modules?

It was shown in the above discussion that the problem with recursive *make* is that the DAGs are incomplete. It follows that by adding the missing portions, the problems would be resolved without abandoning the existing recursive *make* investment.

- The developer needs to remember to do this. The problems will not affect the developer of the module, it will affect the developers of *other* modules. There is no trigger to remind the developer to do this, other than the ire of fellow developers.
- It is difficult to work out where the changes need to be made. Potentially every `Makefile` in the entire project needs to be examined for possible modifications. Of course, you can wait for your fellow developers to find them for you.
- The include dependencies will be recomputed unnecessarily, or will be interpreted incorrectly. This is because *make* is string based, and thus "." and "../ant" are two different places, even when you are in the `ant` directory. This is of concern when include dependencies are automatically generated − as they are for all large projects.

By making sure that each `Makefile` is complete, you arrive at the point where the `Makefile` for at least one module contains the equivalent of a whole-project `Makefile` (recall that these modules form a single project and are thus inter-connected), and there is no need for the recursion any more.

5. Efficient Makefiles

The central theme of this paper is the *semantic* side-effects of artificially separating a `Makefile` into the pieces necessary to perform a recursive *make*. However, once you have a large number of `Makefiles`, the speed at which *make* can interpret this multitude of files also becomes an issue.

Builds can take "forever" for both these reasons: the traditional fixes for the separated DAG may be building too much *and* your `Makefile` may be inefficient.

5.1. Deferred Evaluation

The text in a `Makefile` must somehow be read from a text file and understood by *make* so that the DAG can be constructed, and the specified actions attached to the edges. This is all kept in memory.

The input language for `Makefiles` is deceptively simple. A crucial distinction that often escapes both novices and experts alike is that *make*'s input language is *text based,* as opposed to token based, as is the case for C or even the Unix shell. *Make* does the very least possible to process input lines and stash them away in memory.

As an example of this, consider the following assignment:

```
OBJ = main.o parse.o
```

Humans read this as the variable OBJ being assigned two filenames "main.o" and "parse.o". But *make* does not see it that way. Instead OBJ is assigned the *string* "main.o parse.o". It gets worse:

```
SRC = main.c parse.c
OBJ = $(SRC:.c=.o)
```

In this case humans expect *make* to assign the same two filenames to OBJ, but *make* actually assigns the string "$(SRC:.c=.o)". This is because it is a *macro* language with deferred evaluation, as opposed to one with variables and immediate evaluation.

If this does not seem too problematic, consider the following `Makefile`:

```
SRC = $(shell echo 'Ouch!' \
  1>&2 ; echo *.[cy])
OBJ = \
  $(patsubst %.c,%.o,\
    $(filter %.c,$(SRC))) \
  $(patsubst %.y,%.o,\
    $(filter %.y,$(SRC)))
test: $(OBJ)
  $(CC) -o $@ $(OBJ)
```

How many times will the shell command be executed? **Ouch!** It will be executed *twice* just to construct the DAG, and a further *two* times if the rule needs to be executed.

If this shell command does anything complex or time consuming (and it usually does) it will take *four* times longer than you thought.

But it is worth looking at the other portions of that OBJ macro. Each time it is named, a huge amount of processing is performed:

- The argument to *shell* is a single string (all built-in-functions take a single string argument). The string is executed in a sub-shell, and the standard output of this command is read back in, translating newlines into spaces. The result is a single string.
- The argument to *filter* is a single string. This argument is broken into two strings at the first comma. These two strings are then each broken into sub-strings separated by spaces. The first set are the patterns, the second set are the filenames. Then, for each of the pattern sub-strings, if a filename sub-string matches it, that filename is included in the output. Once all of the output has been found, it is re-assembled into a single space-separated string.
- The argument to *patsubst* is a single string. This argument is broken into three strings at the first and second commas. The third string is then broken into sub-strings separated by spaces, these are the filenames. Then, for each of the filenames which match the first string it is substituted according to the second string. If a filename does not match, it is passed through unchanged. Once all of the output has been generated, it is re-assembled into a single space-separated string.

Notice how many times those strings are disassembled and re-assembled. Notice how many ways that happens. *This is slow.* The example here names just two files but consider how inefficient this would be for 1000 files. Doing it *four* times becomes decidedly inefficient.

If you are using a dumb *make* that has no substitutions and no built-in functions, this cannot bite you. But a modern *make* has lots of built-in functions and can even invoke shell commands on-the-fly. The semantics of *make*'s text manipulation is such that string manipulation in *make* is very CPU intensive, compared to performing the same string manipulations in C or AWK.

5.2. Immediate Evaluation

Modern *make* implementations have an immediate evaluation "`:=`" assignment operator. The above example can be re-written as

```
SRC := $(shell echo 'Ouch!' \
  1>&2 ; echo *.[cy])
OBJ := \
  $(patsubst %.c,%.o,\
    $(filter %.c,$(SRC))) \
  $(patsubst %.y,%.o,\
    $(filter %.y,$(SRC)))
test: $(OBJ)
  $(CC) -o $@ $(OBJ)
```

Note that *both* assignments are immediate evaluation assignments. If the first were not, the shell command would always be executed twice. If the second were not, the expensive substitutions would be performed at least twice and possibly four times.

As a rule of thumb: always use immediate evaluation assignment unless you knowingly want deferred evaluation.

5.3. Include Files

Many `Makefiles` perform the same text processing (the filters above, for example) for every single *make* run, but the results of the processing rarely change. Wherever practical, it is more efficient to record the results of the text processing into a file, and have the `Makefile` include this file.

5.4. Dependencies

Try to be careful about include files. They are relatively inexpensive to read, so more rather than less doesn't greatly affect efficiency.

As an example of this, it is first necessary to describe a useful feature of GNU Make: once a `Makefile` has been read in, if any of its included files were out-of-date (or do not yet exist), they are re-built, and then *make* starts again, which has the result that *make* is now working with up-to-date include files. This feature can be exploited to obtain automatic include file dependency tracking for C sources. The obvious way to implement it, however, has a subtle flaw.

```
SRC := $(wildcard *.c)
OBJ := $(SRC:.c=.o)
test: $(OBJ)
  $(CC) -o $@ $(OBJ)
include dependencies
dependencies: $(SRC)
  depend.sh $(CFLAGS) \
    $(SRC) > $@
```

The `depend.sh` script prints lines of the form

file`.o:` *file*`.c` *include*`.h` ...

The most simple implementation of this is to use *GCC,* but you will need to edit the output if you want to avoid dependencies on system include files:

```
#!/bin/sh
gcc -M -MG $* |
sed 's@ /[^ ]*@@g'
```

This implementation of tracking C include dependencies has several serious flaws, but the one most commonly discovered is that the `dependencies` file does not, itself, depend on the C include files. That is, it is not re-built in one of the include files changes. There is no edge in the DAG joining the `dependencies` vertex to any of the include file vertices. If an include file changes to include another file (a nested include), the dependencies will not be recalculated, and potentially the C file will not be recompiled, and thus the program will not be re-built correctly.

A classic build-too-little problem, caused by giving *make* inadequate information, and thus causing it to build an inadequate DAG and reach the wrong conclusion.

The traditional solution is to build too much:

```
SRC := $(wildcard *.c)
OBJ := $(SRC:.c=.o)
test: $(OBJ)
  $(CC) -o $@ $(OBJ)
include dependencies
.PHONY: dependencies
dependencies: $(SRC)
  depend.sh $(CFLAGS) \
    $(SRC) > $@
```

Now, even if the project is completely up-do-date, the dependencies will be re-built. For a large project, this is very wasteful, and can be a major contributor to *make* taking "forever" to work out than nothing needs to be done.

There is a second problem, and that is that if any *one* of the C files changes, *all* of the C files will be re-scanned for include dependencies. This is as inefficient as having a `Makefile` which reads

```
prog: $(SRC)
  $(CC) -o $@ $(SRC)
```

What is needed, in exact analogy to the C case, is to have an intermediate form. This is usually given a "`.d`" suffix. By exploiting the fact that more than one file may be named in an include line, there is no need to "link" all of the "`.d`" files together:

```
SRC := $(wildcard *.c)
OBJ := $(SRC:.c=.o)
test: $(OBJ)
  $(CC) -o $@ $(OBJ)
include $(OBJ:.o=.d)
%.d: %.c
  depend.sh $(CFLAGS) $< > $@
```

This has one more thing to fix: just as the object (`.o`) files depend on the source files and the include files, so do the dependency (`.d`) files. This means tinkering with the `depend.sh` script once more:

```
#!/bin/sh
gcc -M -MG $* |
sed -e 's@ /[^ ]*@@g'   \
 -e 's@^\(.*\)\.o:@\1.d \1.o:@'
```

This method of determining include file dependencies results in the `Makefile` including more files than the original method, but opening files is less expensive than rebuilding all of the dependencies every time. Typically a developer will edit one or two files before re-building; this method will rebuild the *exact* dependency file affected (or more than one, if you edited an include file). On balance, this will use less CPU, and less time.

In the case of a build where nothing needs to be done, *make* will actually do nothing, and would work this out very quickly.

5.5. Multiplier

All of the inefficiencies described in this section compound together. If you do 100 `Makefile` interpretations, once for each module, checking 1000 source files can take a very long time – if the interpretation requires complex processing or performs unnecessary work, or both. A whole project *make*, on the other hand, only needs to interpret a single `Makefile`.

6. Projects *versus* Sand-boxes

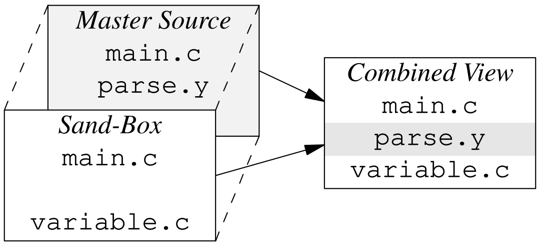
The above discussion assumes that a project resides under a single directory tree, and this is often the ideal. However, the realities of working with large software projects often lead to weird and wonderful directory structures in order to have developers working on different sections of the project without taking complete copies and thereby wasting precious disk space.

It is possible to see the whole-project *make* proposed here as impractical, because it does not match the evolved methods of your development process.

The whole-project *make* proposed here does have an effect on development methods: it can give you cleaner and simpler build environments for your developers. By using *make*'s VPATH feature, it is possible to copy only those files you need to edit into your private work area, often called a *sand-box.*

The simplest explanation of what VPATH does is to make an analogy with the include file search path specified using `-I`*path* options to the C compiler. This set of options describes where to look for files, just as VPATH tells *make* where to look for files.

By using VPATH, it is possible to "stack" the sand-box *on top of* the project master source, so that files in the sand-box take precedence, but it is the union of all the files which *make* uses to perform the build.



In this environment, the sand-box has the same tree structure as the project master source. This allows developers to safely change things across separate modules, *e.g.* if they are changing a module interface. It also allows the sand-box to be physically separate – perhaps on a different disk, or under their home directory. It also allows the project master source to be read-only, if you have (or would like) a rigorous check-in procedure.

Note: in addition to adding a `VPATH` line to your development `Makefile`, you will also need to add `-I` options to the `CFLAGS` macro, so that the C compiler uses the same path as *make* does. This is simply done with a 3-line Makefile in your work area – set a macro, set the VPATH, and then include the Makefile from the project master source.

6.1. Patch

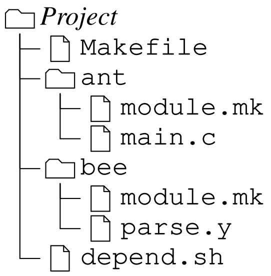
The POSIX semantics of `VPATH` are slightly brain-dead, and GNU Make is POSIX compliant, so the above discussion assumes you have a GNU Make with Paul Smith's VPATH+ patch applied. This may be obtained from `ftp://ftp-.wellfleet.com/netman/psmith/-gmake/`. Here is an extract from the README file:

> Once GNU make finds a target file through VPATH/vpath it changes the target path to the VPATH/vpath name immediately, causing all dependencies of the target to be searched for in the VPATH directory to the exclusion of local files which might be newer.

See the actual README file for a longer explanation.

7. The Big Picture

This section brings together all of the preceding discussion, and presents the example project with its separate modules, but with a whole-project `Makefile`. The directory structure is changed little from the recursive case, except that the deeper `Makefiles` are replaced by module specific include files:



The `Makefile` looks like this:

```
MODULES := ant bee
# look for include files in
#   each of the modules
CFLAGS += $(patsubst %,-I%,\
  $(MODULES))
# extra libraries if required
LIBS :=
# each module will add to this
SRC :=
# include the description for
#   each module
include $(patsubst %,\
    %/module.mk,$(MODULES))
# determine the object files
OBJ :=                     \
  $(patsubst %.c,%.o,      \
    $(filter %.c,$(SRC)))  \
  $(patsubst %.y,%.o,      \
    $(filter %.y,$(SRC)))
# link the program
prog: $(OBJ)
  $(CC) -o $@ $(OBJ) $(LIBS)
# include the C include
#   dependencies
include $(OBJ:.o=.d)
# calculate C include
#   dependencies
%.d: %.c
  depend.sh $(CFLAGS) $< > $@
```

The `ant/module.mk` file looks like:

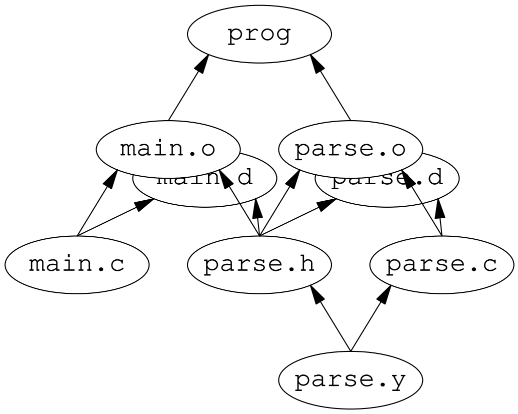
```
SRC += ant/main.c
```

The `bee/module.mk` file looks like:

```
SRC += bee/parse.y
LIBS += -ly
%.c %.h: %.y
  $(YACC) -d %.y
  mv y.tab.c %.c
  mv y.tab.h %.h
```

Notice that the built-in rules are used for the C files, but we need special yacc processing to get the generated `.h` file.

The equivalent DAG of the `Makefile` after all of the includes looks like this:



The vertexes and edges for the include file dependency files are also present as these are important for *make* to function correctly.

8. Literature Survey

How can it be possible that we have been misusing *make* for 20 years? How can it be possible that behavior previously ascribed to *make*'s limitations is in fact a result of misusing it?

The author only started thinking about the ideas presented in this paper when faced with a number of ugly build problems on utterly different projects, but with common symptoms. By stepping back from the individual projects, and closely examining the thing they had in common, *make*, it became possible to see the larger pattern. Most of us are too caught up in the minutiae of just getting the rotten build to work that we don't have time to spare for the big picture. Especially when the item in question "obviously" works, and has done so continuously for the last 20 years.

It is interesting that the problems of recursive *make* are rarely mentioned in the very books Unix programmers rely on for accurate, practical advice.

8.1. The Original Paper

The original *make* paper [1] contains no reference to recursive *make,* let alone any discussion as to the relative merits of whole project *make* over recursive *make.*

It is hardly surprising that the original paper did not discuss recursive *make*, Unix projects at the time usually *did* fit into a single directory.

It may be this which set the "one `Makefile` in every directory" concept so firmly in the collective Unix development mind-set.

8.2. GNU Make

The GNU Make manual [2] contains several pages of material concerning recursive *make,* however its discussion of the merits or otherwise of the technique are limited to the brief statement that

> "This technique is useful when you want to separate makefiles for various subsystems that compose a larger system."

No mention is made of the problems you may encounter.

8.3. Managing Projects with Make

The Nutshell Make book [3] specificly promotes recursive *make* over whole project *make* because

> "The cleanest way to build is to put a separate description file in each directory, and tie them together through a master description file that invokes *make* recursively. While cumbersome, the technique is easier to maintain than a single, enormous file that covers multiple directories." (pp. 65)

This is despite the book's advice only two paragraphs earlier that

> "*make* is happiest when you keep all your files in a single directory." (pp. 64)

Yet the book fails to discuss the contradiction in these two statements, and goes on to describe one of the traditional ways of treating the symptoms of incomplete DAGs caused by recursive *make*.

The book may give us a clue as to why recursive *make* has been used in this way for so many years. Notice how the above quotes confuse the concept of a directory with the concept of a `Makefile`.

This paper suggests a simple change to the mind-set: directory trees, however deep, are places to store files; `Makefiles` are places to describe the relationships between those files, however many.

8.4. BSD Make

The tutorial for BSD Make [4] says nothing at all about recursive *make*, but it is the first the author read which actually described, however briefly, the relationship between a `Makefile` and a DAG (p. 30). There is also a wonderful quote

> "If *make* doesn't do what you expect it to, it's a good chance the `makefile` is wrong." (p. 10)

Which is a pithy summary of the thesis of this paper.

9. Summary

This paper presents a number of related problems, and demonstrates that they are not inherent limitations of *make*, as is commonly believed, but are the result of presenting incorrect information to *make*. This is the ancient *Garbage In, Garbage Out* principle at work. Because *make* can only operate correctly with a complete DAG, the error is in segmenting the `Makefile` into incomplete pieces.

This requires a shift in thinking: directory *trees* are simply a place to hold files, `Makefiles` are a place to remember relationships between files. Do not confuse the two because it is as important to accurately represent the relationships between files in different directories as it is to represent the relationships between files in the same directory. This has the implication that there should be exactly one `Makefile` for a project, but the magnitude of the description can be managed by using a *make* include file in each directory to describe the subset of the project files in that directory.

This paper has shown how a project build and a development build can be equally brief for a whole-project *make*. Given this parity of time, the gains provided by accurate dependencies mean that this process will, in fact, be faster than the recursive *make* case, and more accurate.

9.1. Inter-dependent Projects

In organisations with a strong culture of re-use, implimenting whole-project *make* can present challenges. Rising to these challenges, however, may require looking at the bigger picture.

- A module may be shared between two programs because the programs are closely related. Clearly, the two programs plus the shared module belong to the same project (the module may be self-contained, but the programs are not). The dependencies must be explicitly stated, and changes to the module must result in both programs being recompiled and relinked as appropriate. Combining them all into a single project means that whole-project *make* can accomplish this.
- A module may be shared between two projects because they must inter-operate. Possibly your project is bigger than your current directory structure implies. The dependencies must be explicitly stated, and changes to the module must result in both projects being recompiled and relinked as appropriate. Combining them all into a single project means that whole-project *make* can accomplish this.
- It is the normal case to omit the edges between your project and the operating system or other installed third party tools. So normal that they are ignored in the `Makefiles` in this paper, and they are ignored in the built-in rules of *make* programs.

  Modules shared between your projects may fall into a similar category: if they change, you will deliberately re-build to include their changes, or quietly include their changes whenever the next build may happen. In either case, you do not explicitly state the dependencies, and whole-project *make* does not apply.
- Re-use may be better served if the module were used as a template, and divergence between two projects is seen as normal. Duplicating the module in each project allows the dependencies to be explicity stated, but requires additional effort if maintenance is required to the common portion.

How to structure dependencies in a strong re-use environment thus becomes an exercize in *risk management*. What is the danger that omitting chunks of the DAG will harm your projects? How vital is it to rebuild if a module changes? What are the consequences of *not* rebuilding immediately? How can you tell when a rebuild is necessary if the dependencies are not explicitly stated? What are the consequences of forgetting to rebuild?

9.2. Return On Investment

Some of the techniques presented in this paper will improve the speed of your builds, even if you continue to use recursive *make*. These are not the focus of this paper, merely a useful detour.

The focus of this paper is that you will get more accurate builds of your project if you use whole-project *make* rather than recursive *make*.

- The time for *make* to work out that nothing needs to be done will not be more, and will often be less.
- The size and complexity of the total `Makefile` input will not be more, and will often be less.
- The difficulty of maintaining the `Makefile` will not be more, and will often be less.

The disadvantages of using whole-project *make* over recursive *make* are often un-measured. How much time is spent figuring out why *make* did something unexpected? How much time is spent figuring out that *make* **did** something unexpected? How much time is spent tinkering with the build process? These activities are often though of as ‘‘normal’’ development overheads.

Building your project is a fundamental activity, if it is performing poorly so is development, debugging and testing. Building your project needs to be so simple the newest recruit can do it immediately with only a single page of instructions. Building your project needs to be so simple that it rarely needs any development effort at all. Is your build process this simple?

10. References

1. Stuart I. Feldman, “Make − A Program for Maintaining Computer Programs,” Computing Science Technical Report 57, Bell Laboratories (Aug 1978).
2. Richard M. Stallman and Roland McGrath, *GNU Make: A Program for Directing Recompilation,* Free Software Foundation, Inc., Cambridge, Massachusetts, USA (Jul 1993).
3. Steve Talbott, *Managing Projects with Make, 2nd Ed.,* Nutshell, O’Reilly & Associates, Inc., Newton, MA, USA (1991). ISBN: 0-937175-18-8.
4. Adam de Boor, *PMake − A Tutorial,* University of California, Berkeley, Berkeley, CA, USA (Jul 1988).