Maintaining Files on Plan 9 with Mk

Andrew G. Hume
andrew@research.att.com
Bob Flandrena
bobf@plan9.bell-labs.com

ABSTRACT

`Mk` is a tool for describing and maintaining dependencies between files. It is similar to the UNIX program `make`, but provides several new capabilities. `Mk`'s flexible rule specifications, implied dependency derivation, and parallel execution of maintenance actions are well-suited to the Plan 9 environment. Almost all Plan 9 maintenance procedures are automated using `mk`.

1. Introduction

`Mk` performs the same task as `make` [Feld79] but is different in many ways. This document describes `mk`'s capabilities and discusses how they are used in Plan 9. There are occasional references to differences between `mk` and `make`, but an appendix summarizes the details.

`Mk` works well in the Plan 9 environment. It supports program construction in a heterogeneous environment and exploits the power of multiprocessors by executing maintenance actions in parallel. It interacts seamlessly with the Plan 9 command interpreter `rc` and accepts pattern-based dependency specifications that are not limited to describing rules for program construction. The result is a tool that is flexible enough to perform many maintenance tasks including database maintenance, hardware design, and document production.

This document begins by discussing the syntax of the input file, the pattern matching capabilities, and the special rules for maintaining archives. A brief description of `mk`'s algorithm for deriving dependencies is followed by a discussion of the conventions used to resolve ambiguous specifications. The final sections describe parallel execution and special features.

An earlier paper [Hume87] provides a detailed discussion of `mk`'s design.

2. The `Mkfile`

`Mk` reads a file describing relationships among files and executes commands to bring the files up to date. The specification file, called a `mkfile`, contains three types of statements: assignments, includes, and rules. Assignment and include statements are similar to those in C. Rules specify dependencies between a *target* and its *prerequisites*. When the target and prerequisites are files, their modification times determine if they are out of date. Rules often contain a *recipe*, an *rc*(1) script that produces the target from the prerequisites.

This simple `mkfile` produces an executable from a C source file:

```
CC=pcc
f1: f1.c
	$CC -o f1 f1.c
```

The first line assigns the name of the portable ANSI/POSIX compiler to the `mk` variable `CC`; subsequent references of the form `$CC` select this compiler. The only rule specifies a dependence between the target file `f1` and the prerequisite file `f1.c`. If the target does not exist or if the prerequisite has been modified more recently than the target, `mk` passes the recipe to `rc` for execution. Here, `f1.c` is compiled and loaded to produce `f1`.

The native Plan 9 environment requires executables for all architectures, not only the current one. The Plan 9 version of the same `mkfile` looks like:

```
</$objtype/mkfile
CFLAGS=-w

f1: f1.$O
    $LD $LDFLAGS -o f1 f1.$O
f1.$O:  f1.c
    $CC $CFLAGS f1.c
```

The first line is an include statement that replaces itself with the contents of the file `/$objtype/mkfile`. The variable `$objtype` is inherited from the environment and contains the name of the target architecture. The prototype `mkfile` for that architecture defines architecture-specific variables: `CC` and `LD` are the names of the compiler and loader, `O` is the code character of the architecture. The rules compile the source file into an object file and invoke the loader to produce `f1`. Invoking `mk` from the command line as follows

```
% objtype=sparc mk
kc -w f1.c
kl $LDFLAGS -o f1 f1.k
%
```

produces the `sparc` executable of program `f1` regardless of the current architecture type.

We can extend the `mkfile` to build two programs:

```
</$objtype/mkfile
CFLAGS=-w
ALL=f1 f2

all:V:  $ALL

f1: f1.$O
    $LD $LDFLAGS -o f1 f1.$O
f1.$O:  f1.c
    $CC $CFLAGS f1.c
f2: f2.$O
    $LD $LDFLAGS -o f2 f2.$O
f2.$O:  f2.c
    $CC $CFLAGS f2.c
```

The target `all`, modified by the *attribute* `V`, builds both programs. The attribute identifies `all` as a dummy target that is not related to a file of the same name; its precise effect is explained later. This example describes cascading dependencies: the first target depends on another which depends on a third and so on. Here, individual rules build each program; later we'll see how to do this with a general rule.

3. Variables and the environment

`Mk` does not distinguish between its variables and `rc` variables in the environment. When `mk` starts, it imports each environment variable into a `mk` variable of the same name. Before executing a recipe, `mk` exports all variables, including those inherited from the environment, to the environment in which `rc` executes the recipe.

There are several ways for a variable to take a value. It can be set with an assignment statement, inherited from the environment, or specified on the command line. `Mk` also maintains several internal variables; these variables are described in *mk*(1). Assignments have the following decreasing order of precedence:

1) Command line assignment
2) Assignment statement
3) Imported from the environment
4) Implicitly set by `mk`

For example, a command line assignment overrides a value imported from the environment.

All variable values are strings. They can be used for pattern matching and comparison but not for arithmetic. A *list* is a string containing several values separated by white space. Each member is handled individually during pattern matching, target selection, and prerequisite evaluation.

A *namelist* is a list produced by transforming the members of an existing list. The transform applies a pattern to each member, replacing each matched string with a new string, much as in the substitute command in *sam*(1) or *ed*(1). The syntax is

```
${var:A%B=C%D}
```

where *var* is a variable. The pattern `A%B` matches a member beginning with the string *A* and ending with the string *B* with any string in between; it behaves like the regular expression `A.*B`. When a member of the *var* list matches this pattern, the string *C* replaces *A*, *D* replaces *B*, and the matched string replaces itself. Any of *A*, *B*, *C*, or *D* may be the empty string. In effect, a namelist is generated by applying the *ed*(1) substitute command

```
s/A(.*)B/C\1D/
```

to each member of a variable.

Namelists are useful for generating a list based on a predictable transformation. For example,

```
SRC=a.c b.c c.c
OBJ=${SRC:%.c=%.v}
```

assigns the list `(a.v b.v c.v)` to `OBJ`. A namelist may be used anywhere a variable is allowed except in a recipe.

Command output is assigned to a variable using the normal `rc` syntax:

```
var=`{rc command}
```

The command executes in an environment populated with previously assigned variables, including those inherited from `mk`'s execution environment. The command may be arbitrarily complex; for example,

```
TARG=`{ls -d *.[cyl] | sed 's/..$//'}
```

assigns a list of the C, yacc, and Alef source files in the current directory, stripped of their suffix, to the variable `TARG`.

4. The include statement

The include statement replaces itself with the contents of a file. It is functionally similar to the C `#include` statement but uses a different syntax:

<filename

The contents of the file are evaluated as they are read. An include statement may be used anywhere except in a recipe.

Unlike `make`, `mk` has no built-in rules. Instead, the include statement allows generic rules to be imported from a prototype `mkfile`; most Plan 9 `mkfiles` use this approach [Flan95].

5. Rules

A rule has four elements: targets, prerequisites, attributes, and a recipe. It has the form:

```
targets:attributes:prerequisites
        recipe
```

The first line, containing the targets, attributes, and prerequisites is the *rule header*; it must begin at the left margin. The recipe contains zero or more lines, each of which begins with white space. One or more targets must be specified but the attributes, prerequisites, and recipe are optional. A rule specifies a dependency between the targets and the prerequisites; the recipe brings the targets up to date with the prerequisites. Attributes modify `mk`'s evaluation of a rule.

Normally the target is a file that depends on one or more prerequisite files. `Mk` compares the modification times of each target and each prerequisite; a target is considered out of date when it does not exist or when a prerequisite has been modified more recently. When a target is out of date, `mk` executes the recipe to bring it up to date. When the recipe completes, the modification time of the target is checked and used in later dependency evaluations. If the recipe does not update the target, evaluation continues with the out of date target.

A prerequisite of one rule may be the target of another. When this happens, the rules cascade to define a multi-step procedure. For example, an executable target depends on prerequisite object files, each of which is a target in a rule with a C source file as the prerequisite. `Mk` follows a chain of dependencies until it encounters a prerequisite that is not a target of another rule or it finds a target that is up to date. It then executes the recipes in reverse order to produce the desired target.

The rule header is evaluated when the rule is read. Variables are replaced by their values, namelists are generated, and commands are replaced by their output at this time.

Most attributes modify `mk`'s evaluation of a rule. An attribute is usually a single letter but some are more complicated. This paper only discusses commonly used attributes; see *mk*(1) for a complete list.

The `V` attribute identifies a *virtual* target; that is, a target that is not a file. For example,

```
clean:V:
	rm *.$O $O.out
```

removes executables and compiler intermediate files. The target is virtual because it does not refer to a file named `clean`. Without the attribute, the recipe would not be executed if a file named `clean` existed. The `Q` silences the default printing of a recipe before it is executed. It is useful when the output of a recipe is similar to the recipe:

```
default:QV:
	echo 'No default target; use mk all or mk install'
```

A recipe is an `rc` script. The recipe is optional; when it is missing, the rule is handled specially, as described later. Unlike `make`, `mk` executes recipes without interpretation. After stripping the first white space character from each line it passes the entire recipe to `rc` on standard input. Since `mk` does not interpret a recipe, escape conventions are exactly those of `rc`. Scripts for `awk` and `sed` commands can be embedded just as they would be specified from the command line. `Mk` invokes `rc` with the `-e` flag, which causes `rc` to stop if any command in the recipe exits with a non-zero status; the `E` attribute overrides this behavior and allows `rc` to continue executing in the face of errors. Before a recipe is executed, variables are exported to the environment where `rc` can see them. Recipe commands may not read from standard input because `mk` uses it internally.

References to a variable can yield different values depending on the location of the reference in the `mkfile`. `Mk` resolves variable references in assignment statements and rule headers when the statement is read. Variable references in recipes are evaluated by `rc` when the recipe is executed; this happens after the entire `mkfile` has been read. The value of a variable in a recipe is the last value assigned in the file. For example,

```
STRING=all

all:VQ:
	echo $STRING
STRING=none
```

produces the message `none`. A variable assignment in a recipe does not affect the value of the variable in the `mkfile` for two reasons. First, `mk` does not import values from the environment when a recipe completes; one recipe cannot pass a value through the environment to another recipe. Second, no recipe is executed until `mk` has completed its evaluation, so even if a variable were changed, it would not affect the dependency evaluation.

6. Metarules

A *metarule* is a rule based on a pattern. The pattern selects a class of target and identifies related prerequisites. `Mk` metarules may select targets and prerequisites based on any criterion that can be described by a pattern, not just the suffix transformations associated with program construction.

`Mk` has two types of patterns: *intrinsic* patterns or regular expressions conforming to the syntax of *regexp*(6). The intrinsic patterns are shorthand for common regular expressions. The intrinsic pattern `%` matches one or more of anything; it is equivalent to the regular expression `` `.+' ``. The other intrinsic pattern, `&`, matches one or more of any characters except `` `/' `` and `` `.' ``. It matches a portion of a path and is equivalent to the regular expression `` `[^./]+' ``. An intrinsic pattern in a prerequisite references the string matched by the same intrinsic pattern in the target. For example, the rule

```
%.v:	%.c
```

says that a file ending in `.v` depends on a file of the same name with a `.c` suffix: `foo.v` depends on `foo.c`, `bar.v` depends on `bar.c`, and so on. The string matched by an intrinsic pattern in the target is supplied to the recipe in the variable `$stem`. Thus the rule

```
%.$O:   %.c
    $CC $CFLAGS $stem.c
```

creates an object file for the target architecture from a similarly named C source file. If several object files are out of date, the rule is applied repeatedly and `$stem` refers to each file in turn. Since there is only one `stem` variable, there can only be one `%` or `&` pattern in a target; the pattern `%-%.c` is illegal.

Metarules simplify the `mkfile` for building programs `f1` and `f2`:

```
</$objtype/mkfile
CFLAGS=-w
ALL=f1 f2

all:V:  $ALL

%:  %.$O
    $LD -o $target $prereq
%.$O:   %.c
    $CC $CFLAGS $stem.c
clean:V:
    rm -f $ALL *.[$OS]
```

(The variable `$OS` is a list of code characters for all architectures.) Here, metarules specify compile and load steps for all files. The loader rule relies on two internal variables set by `mk` during evaluation of the rule: `$target` is the name of the target and `$prereq` the name of all prerequisites. Metarules allow this `mkfile` to be easily extended; a new program is supported by adding its name to the third line.

A regular expression metarule must have an `R` attribute. Prerequisites may reference matching substrings in the target using the form `\`*n* where *n* is a digit from 1 to 9 specifying the *n*th parenthesized subexpression. In a recipe, `$stem`*n* is the equivalent reference. For example, a compile rule could be specified using regular expressions:

```
(.+)\.$O:R: \1.c
    $CC $CFLAGS $stem1.c
```

Here, `\1` and `$stem1` refer to the name of the target object file without the suffix. The variable `$stem` associated with an intrinsic pattern is undefined in a regular expression metarule.

7. Archives

`Mk` provides a special mechanism for maintaining an archive. An archive member is referenced using the form *lib*(*file*) where *lib* is the name of the archive and *file* is the name of the member. Two rules define the dependency between an object file and its membership in an archive:

```
$LIB(foo.v):N:  foo.v
$LIB:   $LIB(foo.v)
    ar rv $LIB foo.v
```

The first rule establishes a dependency between the archive member and the object file. Normally, `mk` detects an error when a target does not exist and the rule contains no recipe; the `N` attribute overrides this behavior because the subsequent rule updates the member. The second rule establishes the dependency between the member and the archive; its recipe inserts the member into the archive. This two-step specification allows the archive to represent the state of its members. Other rules can then specify the archive as a prerequisite instead of listing each member.

A metarule generalizes library maintenance:

```
LIB=lib.a
OBJS=etoa.$O atoe.$O ebcdic.$O

$LIB(%):N:  %
$LIB:   ${OBJS:%=$LIB(%)}
    ar rv $LIB $OBJS
```

The namelist prerequisite of the `$LIB` target generates archive member names for each object file name; for example, `etoa.$O` becomes `lib.a(etoa.$O)`. This formulation always updates all members. This is acceptable for a small archive, but may be slow for a big one. The rule

```
$LIB:	${OBJS:%=$LIB(%)}
	ar rv $LIB `{membername $newprereq}
```

only updates out of date object files. The internal variable `$newprereq` contains the names of the out of date prerequisites. The `rc` script `membername` transforms an archive member specification into a file name: it translates `lib.a(etoa.$O)` into `etoa.$O`.

The `mkfile`

```
</$objtype/mkfile
LIB=lib.a
OBJS=etoa.$O atoe.$O ebcdic.$O

prog:	main.$O $LIB
	$LD -o $target $prereq

$LIB(%):N:	%
$LIB:	${OBJS:%=$LIB(%)}
	ar rv $LIB $OBJS
```

builds a program by loading it with a library.

8. Evaluation algorithm

For each target of interest, `mk` uses the rules in a `mkfile` to build a data structure called a dependency graph. The nodes of the graph represent targets and prerequisites; a directed arc from one node to another indicates that the file associated with the first node depends on the file associated with the second. When the `mkfile` has been completely read, the graph is analyzed. In the first step, implied dependencies are resolved by computing the *transitive closure* of the graph. This calculation extends the graph to include all targets that are potentially derivable from the rules in the `mkfile`. Next the graph is checked for cycles; `make` accepts cyclic dependencies, but `mk` does not allow them. Subsequent steps prune subgraphs that are irrelevant for producing the desired target and verify that there is only one way to build it. The recipes associated with the nodes on the longest path between the target and an out of date prerequisite are then executed in reverse order.

The transitive closure calculation is sensitive to metarules; the patterns often select many potential targets and cause the graph to grow rapidly. Fortunately, dependencies associated with the desired target usually form a small part of the graph, so, after pruning, analysis is tractable. For example, the rules

```
%:	x.%
	recipe1
x.%:	%.k
	recipe2
%.k:	%.f
	recipe3
```

produce a graph with four nodes for each file in the current directory. If the desired target is `foo`, `mk` detects the dependency between it and the original file `foo.f` through intermediate dependencies on `foo.k` and `x.foo`. Nodes associated with other files are deleted during pruning because they are irrelevant to the production of `foo`.

`Mk` avoids infinite cycles by evaluating each metarule once. Thus, the rule

```
%:	%.z
	cp $prereq $prereq.z
```

copies the prerequisite file once.

9. Conventions for evaluating rules

There must be only one way to build each target. However, during evaluation metarule patterns often select potential targets that conflict with the targets of other rules. `Mk` uses several conventions to resolve ambiguities and to select the proper dependencies.

When a target selects more than one rule, `mk` chooses a regular rule over a metarule. For example, the `mkfile`

```
</$objtype/mkfile
CFLAGS=-w
FILES=f1.$O f2.$O f3.$O

prog:   $FILES
    $LD -o $target $prereq

%.$O:   %.c
    $CC $CFLAGS $stem.c

f2.$O:  f2.c
    $CC f2.c
```

contains two rules that could build `f2.$O`. `Mk` selects the last rule because its target, `f2.$O`, is explicitly specified, while the `%.$O` rule is a metarule. In effect, the explicit rule for `f2.$O` overrides the general rule for building object files from C source files.

When a rule has a target and prerequisites but no recipe, those prerequisites are added to all other rules with recipes that have the same target. All prerequisites, regardless of where they were specified, are available in the recipe in variable `$prereq`. For example, in

```
</$objtype/mkfile
CFLAGS=-w
FILES=f1.$O f2.$O f3.$O

prog:   $FILES
    $LD -o $target $prereq

%.$O:   hdr.h

%.$O:   %.c
    $CC $CFLAGS $stem.c
```

the second rule adds `hdr.h` as a prerequisite of the compile metarule; an object file produced from a C source file depends on `hdr.h` as well as the source file. Notice that the recipe of the compile rule uses `$stem.c` instead of `$prereq` because the latter specification would attempt to compile `hdr.h`.

When a target is virtual and there is no other rule with the same target, `mk` evaluates each prerequisite. For example, adding the rule

```
all:V:  prog
```

to the preceding example builds the executable when either `prog` or `all` is the specified target. In effect, the `all` target is an alias for `prog`.

When two rules have identical rule headers and both have recipes, the later rule replaces the former one. For example, if a file named `mkrules` contains

```
$O.out: $OFILES
    $LD $LFLAGS $OFILES
%.$O:   %.c
    $CC $CFLAGS $stem.c
```

the `mkfile`

```
OFILES=f1.$O f2.$O f3.$O

<mkrules

$O.out: $OFILES
    $LD $LFLAGS -l $OFILES -lbio -lc
```

overrides the general loader rule with a special rule using a non-standard library search sequence. A rule is neutralized by overriding it with a rule with a null recipe:

```
<mkrules

$O.out:Q:   $OFILES
    ;
```

The `Q` attribute suppresses the printing of the semicolon.

When a rule has no prerequisites, the recipe is executed only when the target does not exist. For example,

```
marker:
    touch $target
```

defines a rule to manage a marker file. If the file exists, it is considered up to date regardless of its modification time. When a virtual target has no prerequisites the recipe is always executed. The `clean` rule is of this type:

```
clean:V:
    rm -f [$OS].out *.[$OS]
```

When a rule without prerequisites has multiple targets, the extra targets are aliases for the rule. For example, in

```
clean tidy nuke:V:
    rm -f [$OS].out *.[$OS]
```

the rule can be invoked by any of three names. The first rule in a `mkfile` is handled specially: when `mk` is invoked without a command line target all targets of the first non-metarule are built. When that rule has multiple targets, the recipe is executed once for each target; normally, the recipe of a rule with multiple targets is only executed once.

A rule applies to a target only when its prerequisites exist or can be derived. More than one rule may have the same target as long as only one rule with a recipe remains applicable after the dependency evaluation completes. For example, consider a program built from C and assembler source files. Two rules produce object files:

```
%.$O:   %.c
    $CC $CFLAGS $stem.c
%.$O:   %.s
    $AS $AFLAGS $stem.s
```

As long as there are not two source files with names like *foo*`.c` and *foo*`.s`, `mk` can unambiguously select the proper rule. If both files exist, there are ambiguous rules to produce *foo*`.$O`, and `mk` exits with an error message.

In Plan 9, many programs consist of portable code stored in one directory and architecture-specific source stored in another. For example, the `mkfile`

```
</$objtype/mkfile
CFLAGS=-w
FILES=f1.$O f2.$O f3.$O f3.$O

prog:   $FILES
    $LD -o $target $prereq

%.$O:   %.$c
    $CC $CFLAGS $stem.c

%.$O:   ../port/%.c
    $CC $CFLAGS ../port/$stem.c
```

builds the program named `prog` using portable code in directory `../port` and architecture-specific code in the current directory. As long as the names of the C source files in `../port` do not conflict with the names of files in the current directory, `mk` selects the appropriate rule. If like-named files exist in both directories, the specification is ambiguous and results in an error. An explicit target resolves the ambiguity. For example, adding the rule

```
f2.$O:  f2.c
    $CC $CFLAGS $f2.c
```

to the previous `mkfile` uses the architecture-specific version of `f2.c` instead of the portable one. Here, the explicit rule documents which of the like-named source files is used to build the program.

`Mk`'s heuristics can produce unintended results when rules are not carefully specified. For example, the rules that build object files from C or assembler source files

```
%.$O:   %.c
    $CC $CFLAGS $stem.c
%.$O:   %.s
    $AS $AFLAGS $stem.s
```

illustrate a subtle pratfall. Adding a header file dependency to the compile rule

```
%.$O:   %.c hdr.h
    $CC $CFLAGS $stem.c
```

produces the error message

```
don't know how to make 'file.c'
```

when *file*.s is an assembler source file. This occurs because *file*`.s` satisfies the assemble rule and `hdr.h` satisfies the compile rule, so either rule can potentially produce the target. When a prerequisite exists or can be derived, all other prerequisites in that rule header must exist or be derivable; here, the existence of `hdr.h` forces the evaluation of a C source file. Specifying the dependencies in different rules avoids this interpretation:

```
%.$O:   hdr.h
%.$O:   %.c
    $CC $CFLAGS $stem.c
```

Although `hdr.h` is an additional prerequisite of the compile rule, the two rules are evaluated independently and the existence of the C source file is not linked to the existence of the header file. However, this specification describes a different dependency. Originally, only object files derived from C files depended on `hdr.h`; now all object files, including those built from assembler source, depend on the header file.

Metarule patterns should be as restrictive as possible to prevent conflicts with other rules. Consider the `mkfile`

```
</$objtype/mkfile
BIN=/$objtype/bin
PROG=foo

install:V:  $BIN/$PROG

%:  %.c
    $CC $stem.c
    $LD -o $target $stem.$O

$BIN/%: %
    mv $stem $target
```

The first target builds an executable in the local directory; the second installs it in the directory of executables for the architecture. Invoking `mk` with the `install` target produces:

```
mk: ambiguous recipes for /mips/bin/foo:
/mips/bin/foo <-(mkfile:8)- /mips/bin/foo.c <-(mkfile:12)- foo.c
/mips/bin/foo <-(mkfile:12)- foo <-(mkfile:8)- foo.c
```

The prerequisite of the `install` rule, `$BIN/$PROG`, matches both metarules because the `%` pattern matches everything. The `&` pattern restricts the compile rule to files in the current directory and avoids the conflict:

```
&:  &.c
    $CC $stem.c
    $LD -o $target $stem.$O
```

10. Missing intermediates

Mk does not build a missing intermediate file if a target is up to date with the prerequisites of the intermediate. For example, when an executable is up to date with its source file, mk does not compile the source to create a missing object file. The evaluation only applies when a target is considered up to date by pretending that the intermediate exists. Thus, it does not apply when the intermediate is a command line target or when it has no prerequisites.

This capability is useful for maintaining archives. We can modify the archive update recipe to remove object files after they are archived:

```
$LIB(%):N:  %
$LIB:   ${OBJS:%=$LIB(%)}
    names=`{membername $newprereq}
    ar rv $LIB $names
    rm -f $names
```

A subsequent mk does not remake the object files as long as the members of the archive remain up to date with the source files. The -i command line option overrides this behavior and causes all intermediates to be built.

11. Alternative out-of-date determination

Sometimes the modification time is not useful for deciding when a target and prerequisite are out of date. The P attribute replaces the default mechanism with the result of a command. The command immediately follows the attribute and is repeatedly executed with each target and each prerequisite as its arguments; if its exit status is non-zero, they are considered out of date and the recipe is executed. Consider the mkfile

```
foo.ref:Pcmp -s:    foo
    cp $prereq $target
```

The command

```
cmp -s foo.ref foo
```

is executed and if foo.ref differs from foo, the latter file is copied to the former.

12. Parallel processing

When possible, mk executes recipes in parallel. The variable $NPROC specifies the maximum number of simultaneously executing recipes. Normally it is imported from the environment, where the system has set it to the number of available processors. It can be decreased by assigning a new value and can be set to 1 to force single-threaded recipe execution. This is necessary when several targets access a common resource such as a status file or data base. When there is no dependency between targets, mk assumes the recipes can be executed concurrently. Normally, this allows multiple prerequisites to be built simultaneously; for example, the object file prerequisites of a load rule can be produced by compiling the source files in parallel. Mk does not define the order of execution of independent recipes. When the prerequisites of a rule are not independent, the dependencies between them should be specified in a rule or the mkfile should be single-threaded. For example, the archive update rules

```
$LIB(%):N:  %
$LIB:   ${OBJS:%=$LIB(%)}
    ar rv $LIB `{membername $newprereq}
```

compile source files in parallel but update all members of the archive at once. It is a mistake to merge the two rules

```
$LIB(%):    %
    ar rv $LIB $stem
```

because an ar command is executed for every member of the library. Not only is this inefficient, but the archive is updated in parallel, making interference likely.

The $nproc environment variable contains a number associated with the processor executing a recipe. It can be used to create unique names when the recipe may be executing simultaneously on several processors. Other maintenance tools provide mechanisms to control recipe scheduling explicitly [Cmel86], but mk's general rules are sufficient for all but the most unusual cases.

13. Deleting target files on errors

The D attribute causes mk to remove the target file when a recipe terminates prematurely. The error message describing the termination condition warns of the deletion. A partially built file is doubly dangerous: it is not only wrong, but is also considered to be up to date so a subsequent mk will not rebuild it. For example,

```
pic.out:D:  mk.ms
        pic $prereq | tbl | troff -ms > $target
```

produces the message

```
mk: pic mk.ms | ...  : exit status=rc 685: deleting 'pic.out'
```

if any program in the recipe exits with an error status.

14. Unspecified dependencies

The -w command line flag forces the files following the flag to be treated as if they were just modified. We can use this flag with a command that selects files to force a build based on the selection criterion. For example, if the declaration of a global variable named *var* is changed in a header file, all source files that reference it can be rebuilt with the command

```
$ mk -w`{grep -l var *.[cyl]}
```

15. Conclusion

There are many programs related to make, each choosing a different balance between specialization and generality. Mk emphasizes generality but allows customization through its pattern specifications and include facilities.

Plan 9 presents a difficult problem, particularly because of its heterogeneous collection of architectures and languages. Mk's flexible specification language and simple interaction with rc work well in this environment.

As a result, Plan 9 relies on mk to automate almost all maintenance. Tasks as diverse as updating the network data base, producing the manual, or building a release are expressed as mk procedures.

16. References

[Cmel86] R. F. Cmelik, ''Concurrent Make: A Distributed Program in Concurrent C'', AT&T Bell Laboratories Technical Report, 1986.

[Feld79] S. I. Feldman, ''Make a program for maintaining computer programs'', *Software Practice & Experience* , 1979 Vol 9 #4, pp. 255-266.

[Flan95] Bob Flandrena, ''Plan 9 Mkfiles'', this volume.

[Hume87] A. G. Hume, ''Mk: A Successor to Make'', *USENIX Summer Conf. Proc.,* Phoenix, Az.

17. Appendix: Differences between make and mk

The differences between mk and make are:

Make builds targets when it needs them, allowing systematic use of side effects. Mk constructs the entire dependency graph before building any target.

Make supports suffix rules and % metarules. Mk supports % and regular expression metarules. (Older versions of make support only suffix rules.)

Mk performs transitive closure on metarules, make does not.

Make supports cyclic dependencies, mk does not.

Make evaluates recipes one line at a time, replacing variables by their values and executing some commands internally. Mk passes the entire recipe to the shell without interpretation or internal execution.

Make supports parallel execution of single-line recipes when building the prerequisites for specified targets. Mk supports parallel execution of all recipes. (Older versions of make did not support parallel execution.)

`Make` uses special targets (beginning with a period) to indicate special processing. `Mk` uses attributes to modify rule evaluation.

`Mk` supports virtual targets that are independent of the file system.

`Mk` allows non-standard out-of-date determination, `make` does not.

It is usually easy to convert a `makefile` to or from an equivalent `mkfile`.