Programming Assignment 3

Issued: Wednesday 25 April 2007
Due: Monday 7 May 2007, 11:59:59pm

Overview

For this assignment you will implement `smake`, a simplified version of the popular `make` utility. This program will actually be a greatly simplified version of `make`, but it will support many of the most commonly written simple makefiles (that do not use variables).

File Format

For our purposes, a Smakefile will consist only of a set of rules. Blank lines (i.e., those consisting only of whitespace) are completely ignored.

A make rule consists of a target, a set of dependencies, and a set of actions.

```
target : dependency1 ... dependencyN
        action1
        ...
        actionM
```

Each action line must start with a tab (`'\t'`) character. Though this is an annoying fact when writing your own make files, it does simplify parsing the file. Any non-blank line that does not start with a tab is a rule. If a rule is missing its separator (`':'`), then report an error.

Dependences : Each dependency is, conceptually, a file name.

Target : The target also, conceptually, names a file.

Actions : Each action is assumed to be a valid Unix command. These commands are executed when it is determined that the target is out-of-date (see below).

For example,

```
main : main.o other.o echo
        gcc -o main main.o other.o
        echo "Done!"

echo :
        echo "Echo"

main.o : main.c
        gcc -c main.c
        echo "main built"

other.o : other.c
        gcc -c other.c
        echo "other built"
```

This example contains four rules. The rule for `echo` contains no dependencies.

For this assignment, we will ignore variable definitions and uses, any circular dependencies, and multiple rules with the same target (by ignore them, I mean that you need not even check for them, though you are certainly allowed to extend your solution to account for such features).

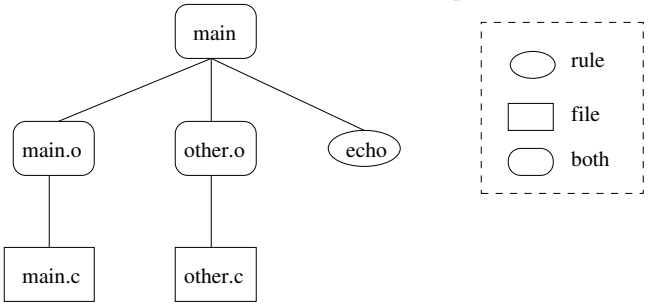
Rule Processing

Rules are applied in an attempt to update the target only if one of its dependencies has been updated. `make` (and therefore `smake`) is based on the concept of files and timestamps as the discussion below reflects.

The processing rules are as follows:

- Check each dependency.
  - If the dependency corresponds to a rule, then recursively apply the rule for that dependency (this means that there is a rule in the Smakefile with this dependency as a target).
  - If the dependency corresponds to a file on the filesystem (but not a rule), then consider that dependency as having been checked.
  - If there is neither a rule nor a file corresponding to the dependency, then report an error.
- Determine if the dependencies were updated. A dependency is considered updated if
  - the timestamp on the corresponding file is newer than that of the target (even if there is a rule for the dependency).
  - the actions for the rule corresponding to the dependency (if there is one) were executed.
- Execute the rule's actions if
  - there are no dependencies.
  - a file with a name matching the target does not exist.
  - any of the dependencies has been updated.

Based on these steps, one can view the processing of a rule as a tree traversal. Consider the following tree rooted at the rule for `main` from the example above.



Target `main` has three dependencies. Two correspond to both rules and files. The third corresponds only to a rule. Each of the rules is recursively applied. The first, `main.o`, depends only on a file (namely, `main.c`). If the timestamp for `main.c` is newer than that of `main.o` (or if `main.o` does not exist), then the actions for `main.o` are executed. Otherwise, nothing happens. The rule for `other.o` works similarly. The rule for `echo` contains no dependencies, so its actions will be executed.

Once the rules for all dependencies have been applied, it can be determined whether the actions for `main` should be executed. If either `main.o` or `other.o` has a newer timestamp (or if their corresponding rules executed their associated actions), then the actions for `main` will be executed. Similarly, if the actions for `echo` were executed (which they always will be in this example), then the actions for `main` will be executed.

Tasks

`smake`

You are to write the `smake` utility. When the program begins executing, it attempts to open a file named `Smakefile` in the current directory. If this file cannot be opened for reading, then execution terminates with an error. If the file can be opened, then processing continues by parsing the file and applying a first rule (see below).

This program can optionally take one command-line argument. If no argument is provided, then the first rule in `Smakefile` is applied. If an argument is provided, then the rule with target equal to the argument is

applied (this rule becomes the root of the tree discussed previously). If there is no matching rule, then execution terminates with an error.

To execute the actions for a rule (when appropriate) you can use the `system` library function. If any executed command fails, report the error and exit immediately with an error code. The actions, when executed, *must* execute in the order given in the action list.

Output: Whenever a command is to be executed, first echo the command to standard out (just as `make` does) and then execute the command.

Useful Functions

You might find the following functions to be of use (this is not to say that you will need them all or that you will not use others)

strtok, strdup, strcat, strcpy, system, stat, lstat

Note

You may **not** assume any limits on line lengths, the number of rules, the number of dependencies per rule, or the number of actions per rule.

Submit

- All source code.
- A Makefile that will build your program.
- A `README` file that specifies how to build your program and that describes anything about your program that you feel I should know during grading.

Grading

Feature	Percentage
Core Functionality	70
Multiple Actions Per Rule	20
Error Handling	10

This is a large-grain grading breakdown. Specific point details are in the testing plan, as implemented in the test execution script, `run.csh`.