Assignment #2

100% Due: April 26, 2016 11:00pm, -5% per day afterwards.

Overview

The primary purpose of this assignment is for you to gain additional experience working in C. Specifically, you will be working with file I/O and dynamic data structures. A secondary purpose, by nature of the project, is to gain additional familiarity with `make`.

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 These commands are executed when it is determined that the target is out-of-date (see below).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).

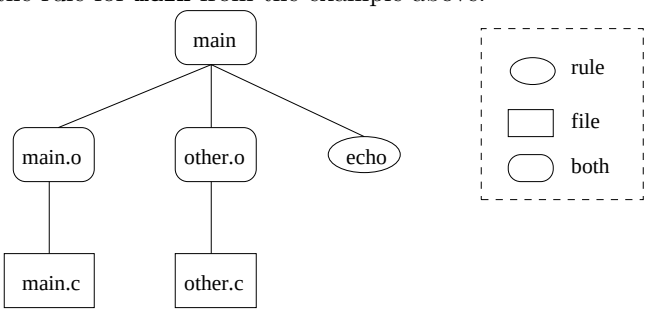
Rule Processing

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.

A rule is processed according to the following steps.

- Validate each dependency.
  For each dependency (`main.o`, `other.o`, and `echo` for the first rule in the example), execute the following steps.
  - Determine the type of dependency.
    A dependency will be one of the following.
    1) The target of another rule. If the dependency is the target of another rule (such as `echo` for the first rule in the example), then recursively apply that rule. Do this even if the dependency also corresponds to an existing file (such as `main.o` for the first rule in the example).
    2) A file. If the dependency is a file (such as `main.c` for the third rule in the example), then see the next step.
    3) Nonexistent. If the dependency does not exist (it is neither a target nor a file), then report an error (this is the "No rule to make target" error reported by `make`).
  - Determine if the dependency has been updated.
    * If the dependency is a file, then it is considered updated if the file's timestamp is more recent than the target's timestamp (this is based on the modification time which is accessible using one of the `stat` related functions).
    * If the dependency is the target of another rule and the actions for that rule were executed as part of the validation step above, then this dependency is considered updated.
    * Note that either or both of the above may apply to a dependency that is both a file and the target of a rule. So a file dependency may be considered updated if the timestamp is more recent or if the rule corresponding to the name was executed.
- Execute this rule's actions if
  - there are no dependencies (as for the second rule in the example).
  - a file with a name matching the target does not exist.
  - any of the dependencies has been updated (determined by the previous step).

Based on these steps, one can view the processing of a rule as a tree traversal (though this does not imply that you must or even should use a tree data structure to implement this). 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, strsep, strdup, strcat, strcpy, system, stat, lstat

Though we will discuss the `stat` functions in more detail, you can take the following code snippet as an example of how to access the modification time for a file.

```
struct stat buf;
if (stat(pathname, &buf))
{
   perror(pathname);
   exit(-1);
}

// buf.st_mtime provides access to the modification time
```

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.

Tips

- Practice iterative development.
- Decide on your data structure(s): implement and test these independently of the program.
- Process the file contents, store the appropriate data in your data structures, and then verify that the data is stored correctly and easily accessible.
- Write and test functions to implement each step.
- Test early and often. If you delay testing until the entire program is written, then you may run into difficulty trying to isolate bugs.

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.

Instructions on submitting will be given later.