CS 352 (Fall 16): System Programming and UNIX

Project #10

`MakeThis`

due at 9pm, Wed 23 Nov 2016

1 Overview

This problem involves writing a C program that implements part the core functionality of the `make` utility. This assignment involves reading in a file specifying dependencies and commands as in `make`; constructing a dependency graph; and then traversing the graph starting with some specified target.

2 Invocation

Your program will be invoked as follows:

```
makeThis [-f <file name>] [<target>]
```

These argument are optional and may appear in any order. The `<file name>` is the name of the makefile. If no file is specified, your program should try to open and use a file named makefileDefault We use this name rather than makefile to avoid conflicting with the makefile you're using to build your project. If no target is specified, the first target defined in your makefile is used.

3 The makefile

The makefile you read will be a simplified version of that which the real make program can deal with. In your makefile the only legal lines will be target definition lines, command lines, and blank lines.

Terminology

A **blank line** is a line that contains only white space (as identified by `isspace()` or the `scanf()` family). Blank lines should be ignored.

A **Command Line** is any nonblank line that starts with a tab.

A **Command** is the string that starts with the first nonwhite character on a command line and ends with the last nonwhite character on that line. Each command is associated with the target that was most recently defined before it.

A **Target Definition Line** is a line that is not a command line or a Command Line that has the form

```
target : dep1 dep2 . . . depn
```

The line might start with white space before the target (though the first char CANNOT be a tab), and there might be 0 or more white spaces before or after the `':'` char. Note also that n might be 0 which means a target definition with no dependencies is legal. It is not legal to have more than one target definition for the same target.

A **Target** is the string that on a Target Definition that starts with the first nonwhite space and continues until either white space or the `':'` is reached. Unlike the real make program, we will only allow one target per target definition.

Dependencies are all the white space separated strings that follow the first ':' on a Target Definition Line.

If a makefile has any lines that don't match the above criteria, it is an illegal file. Also, since each command is associated with the most recently defined target, a file that has a command line before a target line is considered illegal. If the makefile is illegal, your program should print a message to stderr and exit with a status of 1.

Makefile Examples

Here are some examples of good makefiles:

```
file1 : file2 file3
    command to build file1
```

That is about as simple as it gets. Note that a target definition line might have spaces (NOT TAB as the first character!) before the target, it might have 0, 1, or many commands, there might or might not be white space before or after the colon. Blank lines are allowed anywhere. So the following is also a legal file:

```
sam:mark

  mark                  :fred  tom
<tab>command1 to build mark (the line above starts with spaces not a tab)

<tab><tab>command2 to build mark

cindy :

sue:    linda
```

Notice that `cindy` and `sue` have no command, this is legal.

Here are some illegal make files:

```
test me : dep1
    command
```

The above file has more than one target on the same line, and so is illegal.

```
target
    command
```

The file above has no ':' on the target line and thus is illegal.

4 Processing the file

Your program will be invoked with a specification for a file and target. It will read the file, making sure it is legal, and creating a dependency graph based on that file. After that it will do a post order traversal of the graph, starting at the provided target. When each node is "visited" (see below) you will print out the name of the node (the target or dependency) followed by all the commands (if any) associated with that target. To make the commands easier to pick out, print exactly two spaces prior to each command line.

4.1 Dependency Graphs

A dependency graph is a *directed graph* satisfying the following:

1. Each node represents a target or dependency as given in the input makefile file. Thus, for each target definition of the form

   ```
   A : ... B ...
   ```

   there is a node named `A` and a node named `B` in the dependency graph.

2. There is an edge from node `A` to node `B` if `A` "depends on" `B,` i.e., if the input file contains a rule of the form

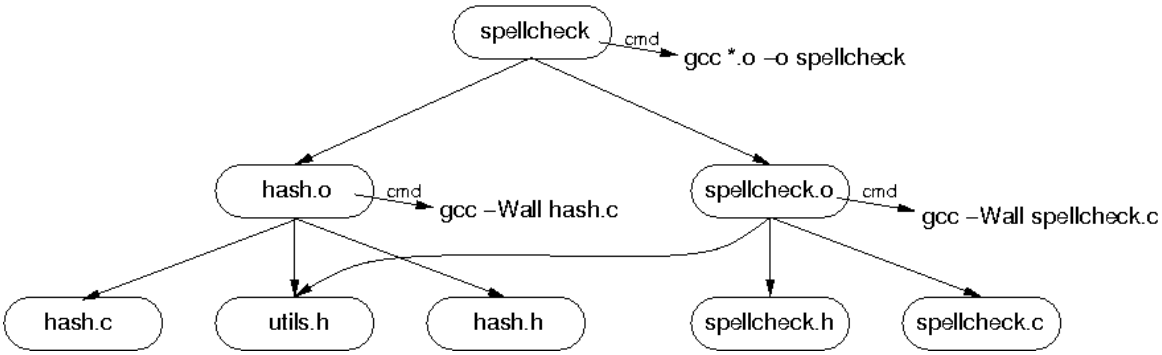
   ```
   A : ... B ...
   ```

The following example illustrates the notion of dependency graphs. Consider the following make file:

```
spellcheck.o : utils.h spellcheck.h spellcheck.c
    gcc -Wall -c spellcheck.c

hash.o : hash.c utils.h hash.h
    gcc -Wall hash.c

spellcheck : hash.o spellcheck.o
    gcc *.o -o spellcheck
```



The ordering on the children of each node in a dependency graph is significant: it reflects the left-to-right ordering on the dependencies specified as part of a rule. For example, the first rule in the example above gives the dependencies of the target `spellcheck.o` in the following left-to-right order:

```
utils.h
spellcheck.h
spellcheck.c
```

The children of the node corresponding to this target in the dependency graph shown above reflect this ordering.

4.2 Post-order Traversal

Let the sequence of children of the node `aTarget` be

$$\texttt{<aChild}_1\texttt{, aChild}_2\texttt{, ..., aChild}_n\texttt{>}$$

, where the ordering on the nodes $\texttt{aChild}_i$ reflects as the ordering specified in the target specification for the rule for `aTarget`, as discussed above. The postorder traversal of the graph starting at node `aTarget` is carried out as follows:

1. For $i = 1, ..., n$ (in that order), carry out a postorder traversal of the graph starting at node $\texttt{aChild}_i$.

2. Process the node `aTarget`. For the purposes of this assignment, "processing a node" simply means printing out its name on one line, followed by the commands (if any) associated with that target.

Example: A post-order traversal of the dependency graph shown above, starting at the root node `spellcheck`, produces the following:

```
hash.c
utils.h
hash.h
hash.o
  gcc -Wall hash.c
spellcheck.h
spellcheck.c
spellcheck.o
  gcc -Wall -c spellcheck.c
spellcheck
  gcc *.o -o spellcheck
```

4.3 Multiple Paths - Only Build a Target Once

Commonly, there is a target which is a dependency of multiple other targets - meaning that there are multiple paths from our start of the graph, to a particular target. Consider this makefile:

```
A: B C
     command to build A from B,C

B: D
     command to build B from D
C: D
     command to build C from D

D:
     command to build D
```

You see that A is dependent on both B and C, and both B and C are dependent on D. Thus, D might be built for two reasons: `A->B->D`, or `A->C->D`. However, you must **only list D once.** In this case, the proper output would be:

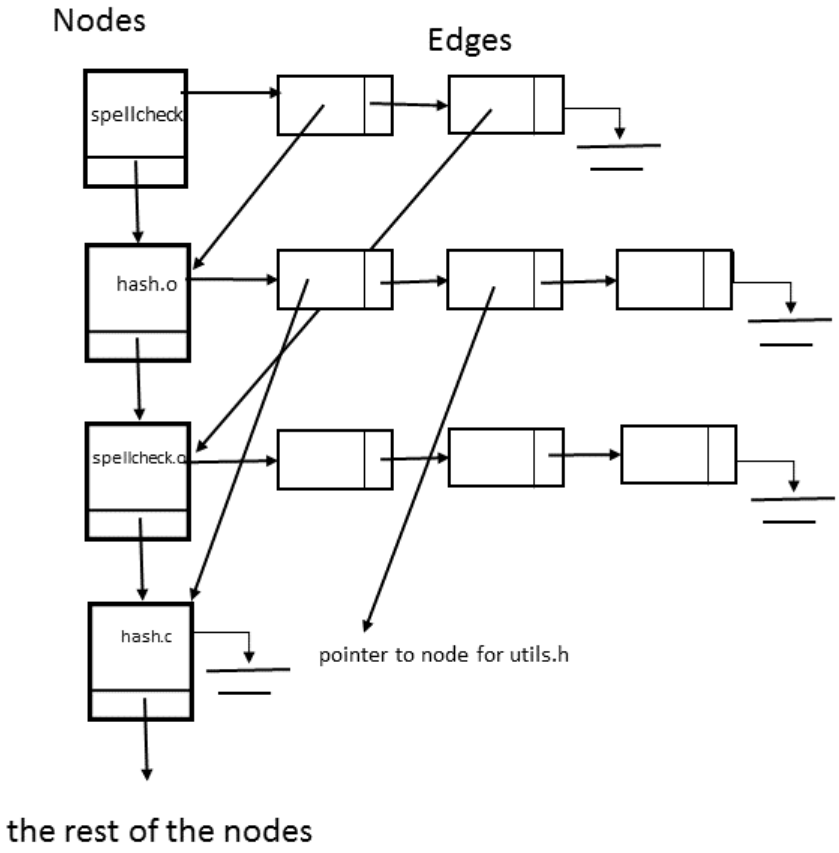
```
D
  command to build D
B
  command to build B
C
  command to build C
A
  command to build A
```

You see that a trivial post-order traversal might have printed D a 2nd time, in between B and C (because D is a dependency of both) - but you must omit this.

Try it out, you'll find that this is how the real `make` utility works!

5 The Data Structure

A good data structure to use for representing a graph is an adjacency list. An adjacency list has a linked list of structures that represent the nodes. In addition to the link to the next node, the structure will contain whatever information is needed for a node. This might include things like the target name, a pointer to a linked list of the commands to build the target, perhaps a "visited" flag to aid in traversing the graph. Also, each node will point to a linked list of "edges" from that node to other nodes. An image of part of this structure for the graph pictured above is shown below. Not all the nodes are shown. Also, even though the image doesn't show it, all edges contain a pointer to a node.



6 Circular Dependencies

A cycle in a dependency graph is called a circular dependency. This can happen when target A depends on target B which in turn depends on target A, e.g. :

```
A : B
    command1
B : C
    command2
C : A
    command3
```

The make program catches such circular dependencies and your program should too. In this case. You should print out an error to standard error when the cycle is reached in the traversal. Your program should not follow the cycle, but should continue the traversal, and finally return a value of 1 upon exit.

Given the file above, and supposing the target was A, your program should output:

```
makeThis: Circular C <- A dependency dropped
C
  command3
B
  command2
A
  command1
```

Where the first line is printed to stderr.

7 Error Conditions

The following are fatal error conditions; your program should print an error message to `stderr`, free used memory, and exit with a status of 1:

- The specified file make file (or the default, if no `-f` parameter is given) does not exist or cannot be opened.
- The makefile contains an illegal line, or mulitiple definitions for the same target, or a command line prior to a target definition.
- The specified target to be built does not exist. (This includes the case where you did not specify a target - but the makefile doesn't contain any targets at all.)

The only non-fatal error in this project is if a circular dependency is encountered when traversing the graph. In that case an error message should be printed, but the program should continue as normal and exit with a status or 1 at the end. You do not need to search the graph for all circular dependencies. In other words, if your file specifies several targets. One target has a circular dependency, but the specified target (the target you are "building") does not, then your program should not produce an error.

8 Multiple Files, Headers, and make

As in earlier assignments, you must break this project up into at least two source files and you must include at least one header file. Also as before, your header file must be structured to protect against multiple inclusion and your makefile must only recompile the files that have changed.

In addition, for this assignment you must structure your makefile to use pattern rules to create only one rule for compiling the object files. (See slides from make deck as well as `/home/cs352/fall16/Assignments/proj08/SOLUTION/lineSort/Makefile` for information).

9 Testing

While it is still a very good idea to use gcov to make sure you've tested your code thoroughly, we will not be grading you on it for this assignment. You may still turn in test directories in the same format as from assignments 7 and 8 if you want to.

10 Grading

10% from Makefile

90% from from code testing

Penalties

-5% - No `#ifndef` protection on the header file

-10% - Only one C file, or no header file.

-10% - Poor style, indentation, variable names

-10% - No file header, or missing header comments for any function (other than `main()`)

-10% - Missing checks of returns codes from standard library functions (such as, but not limited to, `malloc()`)

-20% - Any use of `scanf()` family `%s` specifier without limiting the number of characters read.

11 What You Must Turn In

Turn in the following directory structure using the assignment name `cs352_f16_proj10`:

```
makeThis/
    Makefile
    <at least two C files>
    <at least one header file>
```

12 Standard Requirements

- Your C code should adhere to the coding standards for this class: `http://www.cs.arizona.edu/classes/cs352/fall16/DOCS/coding-standards.html`
- Your programs should indicate whether or not they executed without any problems via their **exit status** - that is, the value returned by `main()`. If you return 0, that means "Normal, no problems." Any other value means that an error occurred.

  In this class, we will always use the value 1 to indicate error; however, outside this class, many programs use many different exit status values - to indicate differnt types of errors.

  In `bash`, you can check the exit status of any command (including your programs) by typing `echo $?` immediately after the program runs (don't run **any** commands in-between).

13 Grade Preview

48 hours before the project is due, we will run the automated grading script on whatever code has been turned in at that time; we'll email you the result. This will give you a chance to see if there are any issues that you've overlooked so far, which will cost you points.

Of course, this grading script will not include the full set of student testcases, which we will use in the grading at the end, but it will give you a reasonable idea of what sort of score you might earn later on.

We will only do this once for this project. If you want to take advantage of this opportunity, then make sure that you have working code (which compiles!) turned in by 48 hours before the due date.