Programming Assignment: fakemake

A `make`-like program for managing precedence constraints.

Due: Friday December 8 by 11:59PM

Overview

A makefile captures the dependencies between different files in a system – usually but not always, files for a software project – and specifies how to generate files or "targets" (e.g., executables, .o files, etc.) from the files on which they depend.

By examining time stamps on files and correctly (and recursively) evaluating the dependencies, the make program can bring a target up to date with a minimal number of operations.

For more information on the `make` program, see the wikipedia entry for it or just type `man make` into a UNIX/Linux shell. This document has an appendix with a real makefile.

In this assignment you will write a program called `fakemake` which mimics the behavior of the UNIX `make` program.

Description

Your program will be called `fakemake` and take a single command line argument – a fake makefile or *fakefile*.

For example, you are given a fake makefile called `ex1` (in the subdirectory `examples`). To start your program using this file, we should be able to simply type:

```
./fakemake ex1
```

Let's just start by looking at this example file (contents listed below).

```
util.h
util.c
stk.h
stk.c
main.c
util.o : util.h util.c
stk.o : stk.h stk.c
app : main.c stk.o util.o
```

Each line of the input file corresponds to a specific "file".[1] There are two types of files:

- **Basic files:** these are files that are depend on no other files. Think of `.c` and `.h` files as in the above example.

  For simplicity, these files are listed one-per-line at the beginning of the file (*before* all composite files - see below).

[1] We use quotes here because your program is a simulation – the "files" only exist inside your program.

- **Target files:** these are files that are "constructed" from other files.

  Think of target files as `.o` files or executables as in the above example. Or from the expand project, the given `.txt` files would be basic and the file generated by expansion would be composite.

  In a correclty formatted input, all target files follow basic files, one per line. Since they are constructed from other files, they *depend* on those files.

  These dependencies are listed on the same line with the target file after a colon `':'`. For simplicity, in a correctly formatted file, there will be whitespace between the target file name and the colon and whitespace between the colon and the dependency list (which technically could be empty).

  See the example above for illustration. Note the following:

  - The dependency list for a target file may include not just basic files, but also other target files. For example, `app` depends on `stk.o` in the sample above.
  - A target file may appear in a dependency list *before* its own line in the file. In other words, the lines for target files can be arbitrarily permuted without changing the specification or legality of the file.

    Below is a such a reording on our initial example.

```
util.h
util.c
stk.h
stk.c
main.c
app : main.c stk.o util.o
stk.o : stk.h stk.c
util.o : util.h util.c
```

Time Stamps/Being Up To Date

Each file in a system has a time stamp – the last time the file was written. Your program will keep track of time stamps as simple integers.

The time stamps determine whether or not a file is "up to date." The rules for a file being up to date are as follows:

- A basic file is always up to date since it doesn't depend on any other files. (A basic file can have it's time stamp chaged however – think a programmer fixing a bug and saving the file).
- A target is up to date if and only if all of the files it depends on are up to date (i.e., this is a recursive definition) **and** the time stamp of the target is greater than or equal to the time stamps of the files it depends on.

When the `make` command is invoked on a particular target, its job is to bring the target up to date if it isn't already; if any of the target's dependencies are out of date, they are brought up to date before bringing the target up to date.

Input File Format

Generically, the file format is as follows:

```
<file1>
<file2>
...
<fileN>
<target1> : <dep1> <dep2> ...
<target2> : <dep1> <dep2> ...
...
<targetK> : <dep1> <dep2> ...
```

In the above specification, there is a total of N "basic" files and K composite target files for a total of $N + K$ vertices.

An example file (with 10 vertices) is as follows.

```
a
b
c
d
e
f
x : a b
y : b c d
z : e f
something : x y z
```

We have intentionally put white space between the target-name and the subsequent colon; and also between the colon and the first dependency. This is to make parsing easier.

When you read in the file, you must make sure of the following:

- No two files (basic or target) have the same name.
- There can be no cycles in the specified graph.

If the input fails either of these tests, your program should print an appropriate error message and terminate. It should *not* enter the interactive loop.

After reading the file, if there are no errors, **all files and targets are initialized to have a time stamp of 0**. (This implies that everything is up-to-date at the beginning – only by performing `touch` operations can things become out of date). The "clock" is intialized to 1.

Then you will enter an interactive mode in which you will support the commands listed below.

`time`**:** This command reports the current value of the clock. **Runtime:** $O(1)$.

`touch <filename>`**:** This command changes the time stamp on the specified filename to the current clock value is and increments the clock. This applies only to "basic" files; not targets with dependencies – an attempt to touch such a file should result in an error message (and the program should continue). **Runtime:** $O(1)$ expected time.

`timestamp <filename>:` This prints the time stamp on the specified file. This is the last time the file – a basic file **or** target – was modified. When your program begins, all files (basic and target) receive time stamps of 0. **Runtime:** $O(1)$ expected time.

`timestamps:` Prints the timestamps of all files. **Runtime:** $O(V)$ time.

`make <target>:` This command takes the target and brings it up to date. Basic files are always up to date. This is basically like the `make` command. See the text above for description. **Runtime:** $O(V + E)$.

`quit:` quits the `fakemake` program.

An important issue is the notion of a "clock". The clock will be used in setting timestamps of the various files. The clock begins at time 1 and is incremented each time the clock value is assigned as the timestamp of a file.

Most of the commands should be clear but the `make` command requires some explanation. When is a file already up to date?

So, when the `make` command is invoked and the target is already up to date, the program will just report that the target is already up to date.

If it is not up to date, it brings each of the files on which it depends up to date (if necessary) and then updates the target. As files get updated, their timestamps get updated, the clock incremented and the actions are reported. As an example based on the above file, you might have something like this (assuming the program has already performed some operations):

```
> touch a
  File 'a' has been modified
> timestamp a
  10
> make something
  making x...done
  y is up to date
  z is up to date
  making something...done
> timestamp x
  11
> timestamp something
  12
> time
  13
```

Important: you will need to be careful to ensure that you don't "update" targets more than necessary. A good example to consider is:

```
a
b : a
c : b
d : b c
```

In this example, if the user invokes `make d`, you need to be careful to *not* make target `b` more than once (if that is necessary).

Your program may be behaviorally correct if you fail to do this, but in the worst case its runtime may be exponential (we'll have a test for this!).

Suggestions

- You will probably find a hash table handy. You may use the `hmap` data structure previously made available.
- There is more than one way to parse an input line. But the `strtok` function can be handy.
- Think Depth-First Search.

Things to remember

- If two files have the same time stamp, that time stamp must be zero. All positive time stamps are unique.
- The clock advances one tick after a new time stamp is assigned to a file (and only when this happens).
- Your program must determine if the input represents an acyclic graph.
- A file's "up-to-date-ness" is a recursive property depending on its dependencies (and their dependencies...)

Resources

hmap: The subdirectory `src` contains an implemention of a hash-map which you are welcome to use.

Think of using a hash-map to keep track of information associated with the various files (which are represented strings right?).

Details are up to you, but here are a couple of ideas:

- Mapping from file names (both basic and target) to an associated integer ID. Each file is really a node in a graph right? It might be handy to internally be able to write code in terms of integer IDs rather than always having to deal with the string representations.

  This idea enables a representation of the dependency graph much along the lines of the adjacency lists we've been seeing in class.

  (Using an array indexed by node ID to get to the edges associated with a node).

- Alternatively, mapping from file names to a list of dependencies. In this approach, the hmap takes on the role of the array in a conventional adjacncy list representation.

Be sure to read the readme file for the hmap code. It explains how you can compile the code depending on what type of "value" you intend to use. (In the first approach above, the value would be an integer; in the second, the value would have to be a generic void pointer).

There are small demo programs for different configurations and the makefile shows how to compile for the different configs.

Sample files: The directory `examples` contains a few sample input files to get you started.

Error Handling

Your program should be 'robust' in its handling of error conditions in the input file and in the subsequent user input.

However, at most 10% of the total points will be determined by test cases that are "well-formed" (syntactically correct).

NOTE: cycle detection is a different story – an input file that has a cycle can still be "well-formed".

In other words, if your program does not correctly detect cycles, you will almost certainly receive more than a 10% deduction!

Submission

You will submit your code in a single archive file.

To compile your program, we should be able to simply type `make fakemake` in your source directory.

Thus, your zipped directory must include a makefile (called either `Makefile` or `makefile`) and all necessary source files to compile your executable. For example, if you used the hmap code, all of those sources must be included and your makefile must compile them accordingly when `make fakemake` is invoked.

Appendix: A Real Makefile

```
# Makefile for a C project
#
# A '#' starts a comment line
# format:
# <target-name>: <list-of-dependencies>
#       <action-to-take-when-to-bring-target-up-to-date>
# Note:  targets may not be actual files (e.g., 'all' below).

all: hello

hello: main.o factorial.o hello.o
        gcc main.o factorial.o hello.o -o hello

main.o: main.c
        gcc -c main.c

factorial.o: factorial.c
        gcc -c factorial.c

hello.o: hello.c
        gcc -c hello.c

clean:
        rm -rf *.o hello
```