# CSc 352 (Fall 2018): Assignment 10

**Due Date:** 11:59PM Wed, Nov 21

The purpose of this assignment is to expand Assignment 9 to function as a simple make program, using simple system commands to check file dates and to run programs.

## General Requirements

1. Your C code should adhere to the coding standards for this class as listed in the Documents section on the Resources tab for Piazza. This includes protecting against buffer overflows whenever you read strings.
2. Your programs should indicate if they executed without any problems via their *exit status*, i.e., the value returned by the program when it terminates:

| Execution | Exit Status |
|---|---|
| Normal, no problems | **0** |
| Error or problem encountered | **1** |

3. Under *bash* you can check the exit status of a command or program ***cmd*** by typing the command "**echo $?**" immediately after the execution of ***cmd***. A program can exit with status *n* by executing "**`exit(`***n***`)`**" anywhere in the program, or by having **`main()`** execute the statement "**`return(`***n***`)`**".
4. Remember your code will be graded on lectura using a grading script. You should test your code on lectura using the **`diff`** command to compare your output to that of the example executable.
5. To get full points your code should compile without warnings or errors when the **`-Wall`** flag is set in **`gcc`**
6. Anytime you input a string you must protect against a buffer overflow. Review slides 82 – 87 of the basic_C deck.
7. You must check the return values to system calls that might fail due to not being able to allocate memory. (e.g. Check that malloc/calloc don't return NULL) getline() is an exception to this rule.
8. Your code must run without errors using valgrind.
9. You must break your code up into at least two source (.c) and one header (.h) files. Your Makefile should create the executable file in such away that only the files that need to are recompiled.

10. **NEW REQUIREMENT: Your Makefile must include a phony target "clean" that removes all the object files in the current directory as well as the executable mymake2.**

# Testing

Example executables of the programs will be made available. You should copy and run these programs on lectura to test your program's output and to answer questions you might have about how the program is supposed to operate. Our class has a home directory on lectura which is:

**`/home/cs352/fall18`**

You all have access to this directory. The example programs will always be in the appropriate **`assignments/assg#/prob#`** subdirectory of this directory. They will have the same name as the assigned program with "ex" added to the start and the capitalization changed to maintain camelback. So, for example, if the assigned program is **`theBigProgram`**, then the example executable will be named **`exTheBigProgram`**. You should use the appropriate UNIX commands to copy these executables to your own directory.

Your programs will be graded by a script. This will include a timeout for all test cases. There must be a timeout or programs that don't terminate will cause the grading script to never finish. This time out will never be less than 10 times the time it takes the example executable to complete with that test input and will usually be much longer than that. If your program takes an exceedingly long time to complete compared to the example code, you may want to think about how to clean up your implementation.

# Makefiles

You will be required to include a Makefile with each program. Running the command:

**`make`** ***`progName`***

should create the executable file *progName*, where *progName* is the program name listed for the problem. The gcc commands in your Makefile that create the object files must include the **`-Wall`** flag. Other than that, the command may have any flags you desire.

Your Makefile should also include a phony target "clean" that removes all the object files in the current directory as well as the executable file created as indicated above. Note this command should not produce error messages even if there are no files to remove.

# Submission Instructions

Your solutions are to be turned in on the host **lectura.cs.arizona.edu**. Since the assignment will be graded by a script, it is important you have the directory structure and the names of the files

exact. Remember that UNIX is case sensitive, so make sure the capitalization is also correct. For all our assignments the directory structure should be as follows: The root directory will be named **assg#**, where # is the number of the current assignment. Inside that directory should be a subdirectory for each problem. These directories will be named **prob#** where # is the number of the problem within the assignment. Inside these directories should be any files required by the problem descriptions.

To submit your solutions, go to the directory containing your assg8 directory and use the following command:

**turnin cs352f18-assg**10 **assg10**

# Problems

# prob1: mymake2

This problem involves extending the **mymake** program from Assignment 9 to have it implement the core functionality of the *make* utility. You will also make the program treat arguments the same way that **make** does.

In Assignment 9 you implemented code to construct and traverse a graph structure reflecting the dependencies specified in an input file. This assignment involves extending the traversal algorithm to check for file existence and last-modified timestamp associated with files, and executing the command associated with a target when it is necessary to rebuild that target.

You will also alter how arguments are handled. **mymake** required 2 arguments. For **mymake2** the arguments will be optional and they can come in either order. To indicate which is which, a -f option flag will proceed the file name.

- **Invocation:** Your program will be compiled to an executable named **mymake2** (see under **Makefile** below). It will be invoked as follows:

  **mymake2** [-f aMakefile] [aTarget]

  *aMakeFile* is is a file specifying dependencies and rules according to the format given here. If no file name is specified, a default file called **myMakefile** should be used. It may not be clear in the browser, but there is white space between the "-f" and the name of the file. *aTarget* is the name of a target appearing in the makefile. If no target is specified, the first target defined in the makefile should be used. This is exactly the same behaivor as **make**. Note that the arguments may appear in either order. The name of the makefile must follow a -f option.

- **Behavior:** An invocation "`mymake2` of your program should behave as follows:

  (i) read in the targets and dependencies specified in the makefile and construct the corresponding data dependency graph; (you have already written the code to do this in assignment 9)

  (ii) traverse this graph using a *postorder traversal*, starting at the node corresponding to the target specified (or the first target defined in the makefile if no target is specifed), recursively (re)build any of the files that the target depends on, and finally (re)build the target itself, as specified in the section labeled "Processing a Target in a Dependency". (This recursive rebuilding amounts to a *post-order traversal*.)

  **Important:** Your program should free all dynamically allocated memory before exiting.

- **Output:** The output produced by your program is the sequence of commands that are executed during the post-order traversal of the dependency graph. (Note that each command that is executed is also printed out to **`stdout`** just before the command is executed.) If no commands are executed, your program should print the line:

  *target* is up to date.

  Where *target* is the name of the target being built.

- **Unix Issues:** You can get the time when a file was last modified using the stat(2) library call: see man 2 stat or man fstat for details. Note that it is your responsibility to allocate a variable of type struct stat and pass a pointer to this variable to stat(), which will then fill in the various fields of the struct with information about the file.
- You can execute a shell command using the library call system(3). See man system for details. Note that you have to declare a variable of type **`struct stat`** and pass a pointer to this variable into the **`stat()`** library call; see **`man 2 stat`** or **`man fstat`** for details.
- **Files:** You should structure your program so that conceptually distinct pieces of the program reside in distinct files. For instance, the code for reading the makefile specifications might be in a different file from the fuctions dealing with graphs. (for example, finding a node, adding a node, traversing the graph, etc.) You should include at least 2 different files of source code and one header file. You can include more if you feel so inclined.
- **Errors:** Error messages should be sensible and informative (use **perror** where necessary) and should be sent to **stderr**.

  The following are all fatal errors and should cause the program to exit immediately with exit status **1**.

  - A file name does not follow a -f argument.
  - -f appears as an argument more than once.
  - More than one target is specified in the arguments.
  - Too many arguments are specified.
  - The input file (either specified or default) cannot be opened for reading.
  - The input file is in an illegal format.
  - The specified target is not defined in the input file.
  - A file the target recursively depends on does not exist and has no rule defining it as a target.
  - A command that is executed is unsuccessful (returns a value other than 0).

# Assumptions and Restrictions

This program obviously builds on your solution to assignment 9. While you may want to alter your code from assignment 9, the work must still be your own.

# Dependency Graphs

A *dependency graph* is a data structure that captures the dependencies between different files as specified in a make file. To keep this discussion specific, we will focus here on rules in makefile2 format; however, the concepts are not specific to this assignment and generalize readily to the full **make** utility.

## 1. Structure

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

Each node represents a *target* or something a target depends on as given in the input mymake2 file. Thus, for each rule of the form

*A* : . . . *B* . . .
\t **cmd1**
\t **cmd2**
there is a node named *A* and a node named *B* in the dependency graph and there is edge from node *A* to node *B* if *A* "*depends on*" *B*.

The sequence of commands (cmd1 and cmd2 in the above example) is associated with the dependency graph node for the target for the rule.

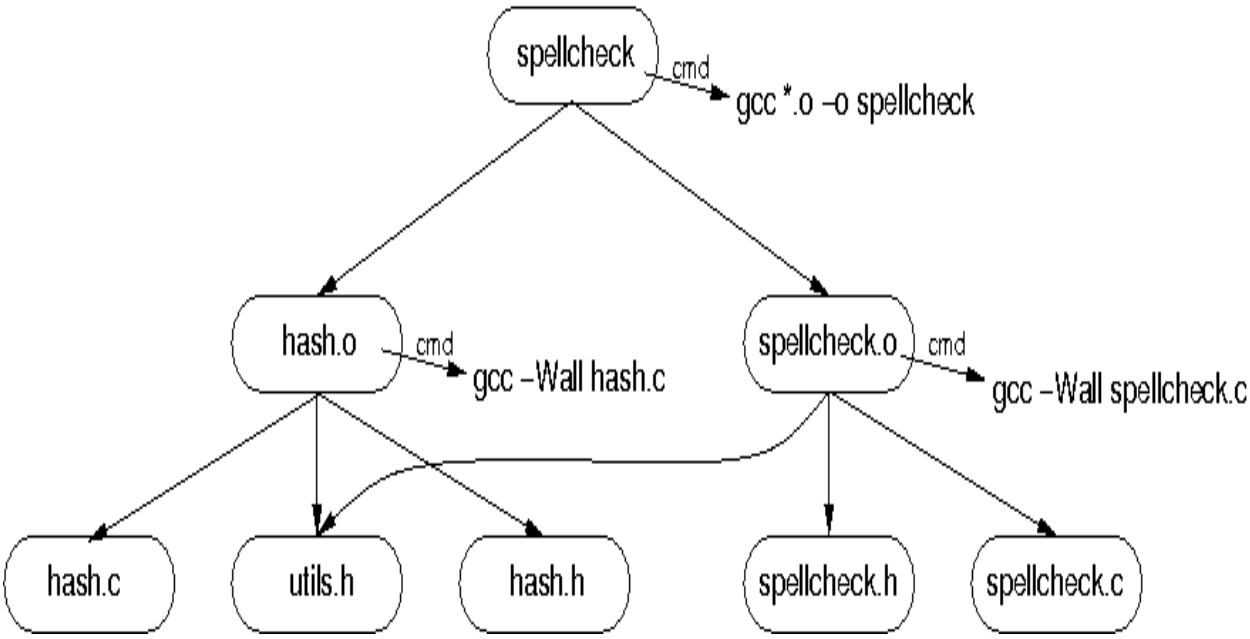
The following example illustrates the notion of dependency graphs. Consider the following mymake2 file:

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

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

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

The corresponding dependency graph is as follows:




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.

# 2. Processing a Target in a Dependency Graph

A target *aTarget* in a dependency graph is processed as follows:

- Suppose that there is a rule with target *aTarget*:

  *aTarget* : $aDep_1$ . . . $aDep_n$

  In this case, we carry out the following actions:

  1. Recursively process each of the targets $aDep_1$ . . .$aDep_n$ (in that order);
  2. If (*a*) *aTarget* does not exist, or (*b*) *aTarget* exists but one (or more) of the files $aDep_i$ that it depends on is newer than *aTarget*, i.e., has a more recent last-modified-time, then the command for the rule for *aTarget* is executed.

At each point of this recursive traversal, your program should print out to **stdout** any command to be executed *immediately before* executing it.

- If there is no rule with target *aTarget*, then the file *aTarget* must exist; if it does not, it is an error.

In other words you should be able to use your code from Assignment 9 to traverse the graph, but instead of printing the name of the node, if that node represents a target, you process it using step 2 above. If it was not a target (i.e. it was a dependency for a target but no command for it was given), then you need to make sure a file with that name exists.

## Further Notes On Processing Commands

- **Circular Dependencies**
  To match the way **make** checks for dates you need to detect circular dependencies and treat them differently from cases where a node has already been visited, but not in a cycle. It is actually not hard to detect cycles. To do so you need to add another mark (or use different values for the same mark) to indicate when a node is "completed". When you first enter your post order traversal function (POT), you should mark the node as "visited". At the very end of the POT you should mark the node as "completed". When looping through a node's dependencies, if you encounter a dependency that has been "visited" but not "completed", then you have detected a cycle. In this case an error message should be printed out and the dependency in question should be ignored. **NOTE that even though you print a message to standard error, the exit status is still 0 in this case. This is an exception to our usual rule.** We make this exception to match the behavior of **make**. The algorithm for this is sketched out below.
- **When to grab modify dates**
  When you grab the modify date will affect whether a command is executed since any command can modify a file and change the date. We will try to mimic **make** in the way we get the dates. In POT, grab the node's modify date after you mark it as visited, but before you go through its dependencies. Then, if the node's commands are run, grab the modify date again after the commands are finished.
- **What to do when no file is found**
  First, we will simplify our check. If stat() does not return a 0, we will assume it is because the file isn't found. There are actually many other reasons stat() might not return a 0, but for simplicity we will ignore them. If the file isn't found and it is not a target, then an error message is printed and execution halts (after freeing memory). If the node is a target, even if it contains no commands that build the file (or even no commands at all), this is NOT an error. This seems strange to me, but it is how **make** works. If the file does not exist, then its commands will be run when that point of the code is reached. If after a target's commands are run the file still does not exist, then any target which has it as a dependency has its commands run.

Here is an algorithm that incorporates what is written above (Note that this depends on some flags being set prior. For instance all the visited flags should initially be false.):

```
POT(n)

  if n.visited then return

  n.visited = true

  set n.fileDate and n.doesExist

  if not n.doesExist
    if n not a target then
      exit with error
    else
      n.mustBuild = true

  for each dependency d of n do

    POT(d)

    if not d.completed (cycle found)
      print err message (but don't change program return status)
    else if not n.mustBuild
      if not d.doesExist or (d.fileDate > n.fileDate)
        n.mustBuild = true

  if n.mustBuild

    run all of n's commands (if any fail, exit with error)

    set n.fileDate and n.doesExist

  n.completed = true
```

Note this is just an algorithm to give what the program should do, not an implementation. You may choose to implement it using different variables. For example, I use one integer field to record both visited and completed.

# 3. Unix Issues

You can get the time when a file was last modified using the **`stat(2)`** library call: see **`man 2 stat`** or **`man fstat`** for details.

Note that it is *your* responsibility to allocate a variable of type **`struct stat`** and pass a pointer to this variable to **`stat()`**, which will then fill in the various fields of the struct with information about the file.

You can execute a shell command using the library call **`system(3)`**. See **`man system`** for details.

# 4. Note on Testing

If you read the man page for 2 stat() then you will see that the modify date is stored in a field of time time_t. What is actually saved is a time in seconds that have elapsed since Jan 1, 1970. The good news is that this means that comparing dates is merely a matter of comparing two numbers (the larger number is more recent). The bad news is that at a resolution of 1 second, mymake2 may not be able to see the more recent file if both were updated within the same second. Once solution is to read at the bottom of the man page how to get the update time in "nano" seconds (I put it in quotes because I'm sure lectura is not that accurate . . . milliseconds maybe). You are not required to do this for this project though. So instead you may want to make your test files work so that at least one second passes before the next targets commands are run. For example, your myMakefile might include lines that look like:

```
A : B C
    touch A
    sleep 1
```

The sleep will cause at least one second difference in time between the time A is updated and when the next targets commands are checked. Note: do NOT put a sleep in your program code. This will slow it down. We will create test files that guarantee if a file is updated then its file date (to the second) will be more recent than other files.