COP5570 Programming Assignment No. 2: A Simplified Make Program

PURPOSE

- Practicing UNIX system calls, and current programming.

DESCRIPTION

In this project, you will implement a simplified make program whose behavior is (almost) the same as the UNIX make utility. The simplified make program organizes a project by performing operations based on the dependencies specified in a makefile. The makefile can have three types of components: macros, target rules, and inference rules. The format for a macro is

```
string1=string2
```

which defines *string*1 to be *string*2. The reference to \$*string*1 or \$(*string*1) will be replaced with string2 during the execution of the make program. The format for a target rule is as follows.

```
Target [target] ... : [prerequisite] [prerequisite] ...
<tab>command1
<tab>command2
...
```

Multiple targets and prerequisites can be specified in the rules. Each of the targets and prerequisites is a string. The format of the commands will be described later. The format of an inference rule is as follows:

```
Target:
<tab>command1
<tab>command2
...
```

The target must be of the form $.s_1$ or $.s_1.s_2$. For an inference rule with the $.s_1$ target, the rule applies to make the target file X from the source file $X.s_1$. For a inference rule with the $.s_1.s_2$ target, the rule applies to make the target file $X.s_2$ from the source file $X.s_1$. Each of the commands associated with a rule can be anything that the shell can execute, including I/O redirection, pipe, and background execution, and the sequential execution of multiple commands.

The simplified make program should support the following features:

- The program should behave like *make*: checking all dependencies (and the timestamps of the related files) in the makefile and performing operations only when needed.
- The default makefiles are *mymake1*, *mymake2*, and *mymake3* in the current directory in order. The default can be overwritten with an "-f" flag. The command line can allow making multiple targets.
- The program should support all three components in the makefile for the UNIX make utility: macros, target rules, and inference rules.
- The makefile should allow comments: everything that follows character '#' is a comment.
- Commands associated with inference rules can use two special symbols: \$@ for the target without suffix and $\$ <$ for the source.

- Commands associated with rules can use defined macros in the form of either $\$string$ or $\$(string)$, as well as the two special symbols (inference rules only).
- Each command line to be supported can be in **one** of the following forms (e.g. you don't need to worry about combining I/O redirection with pipe):
  - a simple command with command line arguments.
  - multiple commands separated by ';'. These commands are to be executed sequentially in order.
  - the 'cd' command (the effect of cd is only on one line of multiple commands).
  - multiple piped commands: *cmd1* | *cmd2* | *cmd3* | *cmd4*. You can limit the number of pipes in a command to be a relatively large number such as 5.
  - a command to be executed in background ('&').
  - a command with redirected I/O ('>' and '<').
- The program should stop executing when the execution of a command failed.
- The paths to search for the commands in a relative path are stored in an environment variable MYMAKEPATH, which has the same format as the $PATH$ variable in *tcsh*.
- The program should allow circular dependencies in the makefile.

DEADLINES AND MATERIALS TO BE HANDED IN

Due date: **Oct. 2**. You should clean and tar your project directory and submit it to the blackboard submission site. In the directory, you should have all the needed files to create the executables of the program. On both *program* and *linprog*, a 'make' command should create the executable. You should also have a README file describing how to compile and run you program and the known bugs in your program. You should also have a makefile that will (1) automatically generate the executable by issuing a 'make' command under the directory, (2) compile the C files with -Wall -ansi -pedantic, (3) clean the directory by issuing 'make clean', and (4) recompile only the related files when a file is modified.

You need to hand in a hard-copy of your programs, makefiles and the README file in the due date when you demonstrate your project.

GRADING POLICY

A program with compiling errors will get 0 point. A program that cannot run any command in the makefile will get 0 point. A program with any 'system' routine in it will get 0 point (the 'system' routine is NOT allowed in the project). Your program should work on both `linprog` and `program` platforms. Any thing that only works on one platform will get 50% of the points.

1. proper README file (5)
2. proper makefile file(for compiling the simplified make utility) (5)
3. using the correct default makefile, the "-f" flag (5)
4. running one rule with one simple command (5)
5. allowing multiple commands in a line (';') (5)
6. allowing the 'cd' command (the changed directory is only in effect in the execution of that line) (5)

7. running multiple piped commands ('|') (5)
8. allowing background execution (5)
9. allowing I/O redirection in a command ('>' and '<') (5)
10. Searching each relative path with paths in MYMAKEPATH (10)
11. supporting macros (5)
12. supporting inference rules, $@ and $ < symbols (5)
13. supporting comments (#) (5)
14. supporting simple dependencies with multiple rules (5)
15. supporting advanced dependencies (e.g. circular dependencies) with multiple rules (5)
16. Overall similar behavior as the UNIX make (10)
17. proper self-guided demo (10)

MISCELLANEOUS

- The implementation of this program will likely be more than 1000 lines of dense code (500- for parsing and dependency checking and 500+ for executing commands in different forms). Start the project as early as you can.
- 20 points for the **second** unknown bug and 10 points for the **second** known bug/unimplemented feature.
- It is normal for system calls to have unexpected behavior. You can either call it a bug or a feature. As a result, you might need to work your way around if that occurs to your program — this is part of the project.
- You can make any assumptions about the format (syntax) of the file as long as the format does not limit the commands' functionality and contradict our specification. Your program is considered correct as long as you can handle the correct inputs.
- There are mainly three modules in this program (1) the module to load the makefile into some internal data structures that can be easily manipulated, (2) the module to check the dependencies and act accordingly (this is likely a recursive routine), and (3) the execution module that runs one line of commands at a time. You can implement the system in the order (1) $\rightarrow$ (2) $\rightarrow$ (3) (use the system routine to run simple commands lines when testing (2)) or (1) $\rightarrow$ (3) $\rightarrow$ (2).
- You should test your make program on all the sample makefiles given in Lecture 2, and make sure that your code work the same as the UNIX make. You will need to add more test cases to test and demonstrate your assignment.
- All programs submitted will be checked by a software plagiarism detection tool.