COP5570 Programming Assignment 2: mymake

PURPOSE

- Practicing UNIX system calls and concurrent programming.

DESCRIPTION

In this project, you will implement a simplified make program, called `mymake`, whose behavior resembles the UNIX `make` utility: you only need to implement the functionality described in this document, but the behavior of your program should be the same as the UNIX `make` for the overlapping functionality.

The `mymake` 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 $X.s_1$.

The command format for `mymake` is as follows. Both options and target are optional.

`./mymake` [options] [target] [options]

`Mymake` supports the following command line options.

- `-f mf`: with this option, the default makefile is replaced with file `mf`. Without this option, the program will search for the default `mymake1.mk`, `mymake2.mk`, `mymake3.mk` in the current directory in order.
- `-p`: with this option, `mymake` should build the rules database from the makefile, output the rules and exit.
- `-k`: with this option, `mymake` should continue execution even when some command fails. Without this option, the default behavior is to report the error and exit when a command fails (the same default behavior as the UNIX `make` utility).

- `-d`: with this option, `mymake` should print debugging information while executing. The debugging information should include the rules applied, and the actions (commands) executed. See the sample executable for more details.
- `-i`: with this option, `mymake` should block the SIGINT signal so that `Ctrl-C` would not have effect on the program. Without this option, the default behavior is that `mymake` will clean up (kill) all of the children that it created and then exit when `Ctrl-C` is typed.
- `-t num`: with this option, `mymake` should run for up to roughly `num` seconds. If the program does not finish in `num` seconds, it should gracefully self-destruct (clean up all of its children and then exit).
- The target is optional. If it is not presented in the command, the default target is the first target in the target rule in the makefile. You can assume there is at least one target rule in the makefile.

The following are some sample correct `mymake` commands:

```
<linprog1 0> ./mymake
<linprog1 1> ./mymake -f makefile
<linprog1 2> ./mymake -f makefile clean
<linprog1 3> ./mymake -f makefile clean -t 20
<linprog1 4> ./mymake -f makefile clean -t 20 -p
<linprog1 5> ./mymake -i -f makefile clean -t 20 -d
<linprog1 6> ./mymake -i -f makefile clean -t 20
<linprog1 7> ./mymake -i clean -k
```

`Mymake` should also 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 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 $@ and $ < (inference rules only).
- Each command line in the action can be in **one** of the following forms (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).
  - Up to 5 piped commands: `cmd1 | cmd2 | cmd3 | cmd4 | cmd5`.
  - a command with redirected I/O ('>' and '<').
- The paths to search for a command (specified as a relative path) are stored in an environment variable `MYPATH`, which has the same format as the `PATH` variable in *tcsh*. Note that you do not need to search for commands specified as an absolute path.

- The program should allow circular dependencies in the makefile.
- You do not need to implement '*' and '∼' and other special symbols in the commands.

DEADLINES AND MATERIALS TO BE HANDED IN

Due date: **February 16**. The project demonstration will be done in the week after.

- Submit all files related to the assignment including makefile, README file, and your self-grading sheet in one tar file to canvas.

  In a project directory that contains your submitted files, a 'make' command should create the executable in linprog. Your README file should describe how to compile and run you program, the known bugs in your program, and how to do your demo. The makefile should (1) automatically generate the executable by issuing a 'make' command under the directory, (2) compile the C/C++ files with '-Wall -ansi -pedantic' or '-Wall -std=cxx -pedantic', (3) clean the directory by issuing 'make clean', and (4) recompile only the related files when a file is modified.

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. You are required to use `execv()` to execute commands. Other exec family routines and the `system()` routine are forbidden: if the *system()* routine or other exec family routines appear in anywhere in the code, the program will automatically get a 0 grade. Your program should work on `linprog`. The total points for this project is 110.

1. proper makefile and README files (8)
2. using the correct default makefile and the "-f" flag (4)
3. running one rule with one simple command (4)
4. allowing multiple commands in a line (';') (4)
5. allowing the 'cd' command (the changed directory is only in effect in the execution of that line) (4)
6. running multiple piped commands ('|') (4)
7. allowing I/O redirection in a command ('>' and '<') (4)
8. Searching each relative path with paths in MYPATH (8)
9. supporting macros (4)
10. supporting inference rules, $@ and $ < symbols (4)
11. supporting comments (#) (4)
12. supporting simple dependencies with multiple rules (4)
13. supporting advanced dependencies (e.g. circular dependencies) with multiple rules (4)
14. with and without `-p` flag (4)
15. with and without `-k` flag (4)

16. with and without `-d` flag (4)
17. with and without `-i` flag (4)
18. with and without `-t` flag (4)
19. graceful exit (no child left behind for Ctrl-C and timeout) (4)
20. overall command line options (4)
21. Overall similar behavior as the UNIX make (12)
22. proper self-guided demo and project submission (10)
23. A project will receive at most 85% of points with the **second** unknown bug
24. -8 points for the **second** known bug/unimplemented feature
25. -5 points for each compiler warning
26. +5 points for being the first one to report a legitimate bug in the sample `partial_mymake` executable.

MISCELLANEOUS

- The implementation of this program should be more than 1000 lines of dense code: 600+ for parsing and dependency checking and enforcing, 600+ for executing commands in different forms, 100+ for signal related and book keeping. Start the project as early as you can.
- 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 makefile as long as the format does not limit the functionality and/or contradict our specification. Your program is considered correct as long as you can handle the correct inputs.
- Besides parsing the command line, there are mainly three modules in this program (1) the module to load the makefile into some internal data structures that can be easily accessed and manipulated, (2) the module to check the dependencies and act accordingly – the make logic (this may be 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 do preliminary test following the instruction of the README file in the project package, make sure that your code works similarly to the UNIX make and/or the sample executable. You will need to add more test cases to test and demonstrate your project.
- All programs submitted will be checked by a software plagiarism detection tool.