Make a Difference – Assignment 3

COMP202 2016 Revised 31 October 2016

Len Hamey, len.hamey@mq.edu.au

This lab is the third of three assignments in COMP202.

Commences: Week 9.

Progress: Weeks 10, 11, 12.

Due: 11:59pm,Friday 11 November (Week 13)

Value: 15% (12% for achievement, 3% for weekly progress)

Rescaling of marks

The marks for this lab have been calibrated based on revised task requirements. There are now four stages in the lab, and each stage is worth 3.0 achievement marks. The automarker still marks each stage out of 2.0, but then it rescales the marks when computing the lab total. This is automatically applied to past achievements as well as future achievements.

Introduction

Make is a system utility for building software. After editing a source file, the programmer runs make on the project. Make examines the modification date-time stamp of the source files and works out what commands need to be executed in order to bring the entire project up to date. Make is more efficient than a shell script that compiles everything, because make won't recompile source files unnecessarily. However, make has some limitations that can, at times, be frustrating.

In this lab assignment, you will develop an implementation of make with a difference. Your make program will still have the basic functionality of make, but it will have some different capabilities that set it apart. Because it is an assignment to be completed in four weeks, there will also be some advanced features of make (particularly the advanced macros) that you do not implement. However, perhaps one day some of the ideas in this lab may be incorporated into make – in that case, your work may help make a difference to make! Even if that never happens, you will learn about system calls, implement some interesting ideas and become an accomplished C systems programmer.

Because the program you will develop is a different make, we have named it `dmake`[1]. When explaining the command line, we will use this name even though you can use any name you wish during your development and testing.

Overview and learning outcomes

The primary goal of this lab is for you to gain experience with Unix system calls. For this reason, we will in some cases specify that your program must use a particular system call or one of a group of related system calls. In other cases, we leave it to you to discover the relevant system calls. In all cases, we expect you to read the Unix manual pages as a primary source of information about the

[1] If you google `dmake`, you will discover an existing version of make with that name. Our `dmake` is not based on that version of make. It is based on GNU make with changes as described in this lab specification.

parameters and usage of the various system calls. You may access the manual pages on the Internet, but the definitive version for our system is on the system itself and is accessed through the `man` command. You are responsible to ensure that you are not misled by documentation that may refer to a different operating system.

The learning outcomes of this lab include the following:

- Ability to write a significant systems program in C
- Ability to understand Unix man pages and use them as an information source
- Ability to choose systems calls appropriate to a task
- Ability to employ systems calls and library calls to achieve a systems task

As with other labs in COMP202, there is an auto-marker for each lab stage that tests your submission, awards a mark and provides feedback. The lab command functions for this lab in the same way as it did for the first lab.

This lab has three progress marks. Progress is awarded for achieving a stage of the lab with a mark of at least 1.5 out of 2.0 (equivalent to 2.25 out of 3.0). The progress mark due dates are available through the lab command. They are

- 11:59pm on Friday 21 October (week 10)
- 11:59pm on Friday 28 October (week 11)
- 11:59pm on Friday 4 November (week 12)

The lab closes at 11:59pm on Friday 11 November (week 13). If you have free extension days remaining at the end of the semester, you may use them to extend the due date of this final lab. This lab is the last opportunity to use your free extension days.

Individual work and information resources

This lab must be your own work. Unlike the previous two labs, all students are attempting the same task for this lab assignment. Because of this, we will be extra diligent to check for plagiarism, whether from other students or from the Internet. In particular, we will use an automated tool to compare source code files and detect copying.

Every submission that you make will be archived by our server. In the event of plagiarism, this will provide evidence to assist us in identifying the circumstances surrounding the plagiarism. We encourage you to submit your program whenever you have made significant changes, even if they are incomplete, as this will provide a record of your development work.

You should not copy code from the Internet, nor should you copy code from other students. You should not copy code or portions of code from the sources of Linux, GNU, or other system implementations of make and related programs – we will obviously check for copying of such sources.

However, you may use resources on the Internet to obtain general information including information about the C language and libraries, information about system calls, and information about the operating system. If you obtain useful information from the Internet, you must include comments at the relevant points in your code acknowledging the source of the information (URL) and briefly describing the key idea(s) that you are using. General information from the Unix manual pages does not require citation, but if there is specific information that influences an important design decision then it would be appropriate to cite the source of the information. Your goal is to help future maintainers of the software to understand what you have done and why.

Incremental development

In this lab you are incrementally developing your implementation of dmake. **The features that you develop for stage 1 should remain in your solution for later stages so that your final solution has all the capabilities of all the stages.** Instead of having different programs for each stage, you will have one program that behaves differently depending on the command line options. You should periodically test your submissions for later stages as submissions for the earlier stages to ensure that you have not broken the earlier features.

Fetching your lab stage

Use the lab command to fetch sample test files for each stage. Refer to this lab specification for the requirements of each lab stage and also to the marking guide that is downloaded with each stage.

Some stages may require precise initialisation of the test files. If this is the case, the downloaded tar file will contain a Makefile. After extracting the files with `tar`, you should `cd` into the extracted folder, use `ls` to see whether there is a Makefile in the file and then, if there is a Makefile, execute `make` to perform the final initialisation steps. E.g.

```
$ cd stage5
$ ls
$ make
```

Submitting your lab solution

Use the lab command to submit your solution. Thanks to the shell, you can use wildcards to submit all the C and H files in your current directory with a single brief command as follows:

```
$ lab -s 3.1 *.c *.h
```

Style notes

Like make, dmake does not require any command-line parameters to operate correctly. Therefore, usage information (brief help) is printed by the option –h. As a partial example:

```
$ dmake –h
Usage: dmake [-h] [-d N]
  -h: Print this usage information
  -z N: Print debugging information for stage N
```

Outline of the stages

1. Read the dmake rules file, parse the rules and print the rules out in debugging format.
2. Process the rules to decide which rules will fire. This involves examining the modification times of files.
3. Extend stage 2 to execute the rules that need to be executed.
4. Check for file differences and avoid unnecessary execution of rules.

[3 marks] Stage 1: Reading and parsing the make file

The first task for `dmake` is to read the make input file and parse it into an internal representation that can be processed in later stages. To verify that you have parsed the input correctly, you are required to dump the parsed information in a specific debugging syntax.

We do not specify the internal representation that you should use, but we encourage good design. Specifically, we encourage simplicity, ease of programming, and ease of future extensions. It is likely that you will use familiar C constructs such as arrays, structs, character strings, and pointers. Your program should contain suitable comments describing the internal representation of the make file. You should also consider *Some Important Comments on Code Style* and *Systems Programming Style* guidelines.

The system `make` command reads its input file from `Makefile` by default. We encourage you to use the system `make` command to manage your source compilations for this project, so we specify the default input file for `dmake` as `Dmakefile` to avoid confusion.

The input file is a text file. Some lines of the input file are empty, some are comments, and the remaining lines are (in this stage) rules[2]. Rules commence with a rule header line that defines targets and dependencies. Each rule contains zero or more commands that are to be executed if the rule fires – i.e. if the targets need to be created or updated.

The syntax of each type of line is as follows. The syntax for `dmake` is similar to `make`, but is not the same.

- Continuation of a line is indicated by a backslash (\) as the last character. Continuation means that the following line is joined to the end of the current line after removing the backslash, and then the result is treated as a single line. There may be multiple lines of continuation. Continuation may be applied to any type of line: command, header or command.

> 👍 Line continuation is an advanced feature that gains only a small mark increment. You can ignore it to achieve progress in stage 1, then add it at a later time.

- Empty lines contain only zero or more white-space characters – such as tabs and/or spaces. Empty lines are ignored completely. However, an empty line after a continuation becomes part of the previous line and ends the continuation.
- Comment lines have the character '#' as the first non-white-space character on the line. (White-space characters are tab and space). Comment lines are completely ignored by the parser.
- Rule header lines commence with a non-white-space character (other than '#') in the first position of the line, and they have a colon (':') somewhere in the line. The contents of dependency lines are file names. The file name(s) appearing before the colon are target(s) – they are the file(s) that the `dmake` rule is designed to create. The file names appearing after the colon are the dependencies - the files that are required in order to execute the command(s) that create the target(s). The file names are separated by tabs and/or spaces.

[2] In a later stage we may introduce additional types of input lines such as macro definitions.

- Command lines commence with one or more white-space characters (e.g. tab or space). Each command line indicates a Unix command that is to be executed. The leading white space should be removed from the command.

For example, consider the following `Dmakefile`:

```
# This is a comment

project: project.c defs.h
  gcc -o project project.c
```

In this simple `dmake` input file, there is a single comment line and an empty line – both are ignored by the parser. There is also one rule. The rule header indicates that the file `project` is the target and it depends on `project.c` and `defs.h`. The command line provides a `gcc` command that could be used to create or update the file `project` from the files `project.c` and `defs.h`. In general, rules may have one or more targets, zero or more dependencies and zero or more commands.

It is useful for a systems program such as `dmake` to have a debug option that reports the internal operation of the program. Indeed, the Linux `make` command has such an option. However, `dmake`'s debug option (`-z`) will behave differently. The debug output from `dmake` will be appropriate to the lab stage specified in the option and specific to the operation of `dmake`.

The download for stage 1 includes several sample `dmake` folders, and corresponding stage 1 debug output. The dmake files are each in their own subdirectory test1, test2, test3, …; the corresponding debug output files are test1.out, test2.out, test3.out, … and the error files are test1.err, test2.err, test3.err, …. The dmake command for each test case is available as test1.cmd, test2.cmd, test3.cmd, … – this shows you the command-line options for each test case. Note that test9 is the case where the Dmakefile is missing and an error response is required.

Please note the following requirements:

1. Use `getopt`(3) (i.e. `getopt` as described in section 3 of the Unix manual) to parse the `dmake` command line.
2. The `dmake` debug option `-z` accepts a parameter which is the stage number. So, for stage 1 debug output the option is either "`-z1`" or "`-z 1`", because `getopt` permits both forms. Clearly, this is different than the debug option for the Unix `make` command. With this option, `dmake` should print the stage 1 debug output and then exit without error.
3. The `dmake` file name option `-f` accepts a parameter which is the input file name. Without this option, `dmake` will read `Dmakefile`. This option is convenient for testing `dmake` with several different input files. Your program will be tested both with and without the `-f` option.
4. Your program output needs to match the sample output files except for differences in white spacing (see below). The syntax of the debug output is intended to be simple, but should require your program to parse the input file and accurately report the information that has been extracted. The sample output captures both standard output and standard error as separate files.
5. It is required that your program parses the input file and produces an internal data structure representing the `dmake` rules. A program that produces the correct output but does not fully parse the input is not considered a correct solution and will hinder your progress in the

future lab stages. Fully parsing the input means: separating the target file names, separating the dependency file names, and removing leading white space from the commands.

6. Syntax errors in the dmake file should be detected and reported to standard error. The error messages are captured in the appropriate sample output files.
7. You may read the dmake file using stdio facilities (such as fopen, fgetc, fgets, etc) but you should ideally handle arbitrarily long lines and arbitrarily large files. However, you can obtain most of the marks by assuming suitable large line length limits and a suitable limit for the number of rules/lines in the input file.

Examples:

```
$ dmake -z1
```

The above command reads `Dmakefile` and prints out the stage 1 debug report for that input file.

```
$ dmake -f Dmakefile2 -z1
```

The above command reads `Dmakefile2` and prints out the stage 1 debug report for that input file.

To test your dmake command, run it with IO redirection to capture the standard output and standard error to a pair of files. You can then compare the output files that you have captured with the expected output. The bash shell syntax for capturing standard output is '>' and the syntax for capturing standard error is '2>'. For example, to run dmake and capture the output to mytest.out and mytest.err:

```
$ dmake -z1 > mytest.out 2> mytest.err
```

Compare your output with the sample output using the `diff` command with the option `-b` to ignore non-significant variations in white space.

```
$ diff -b test1.out mytest.out
$ diff -b test1.err mytest.err
```

Useful system calls and library calls

- getopt is required for parsing the command line. A template dmake.c program is provided which uses this call to parse the –h option, but you need to add other options.
- Standard I/O should be used to read the dmake file: fopen, fclose, fgetc, fgets, etc.
- Standard I/O is used to print the debug output and error messages: printf, fprintf, etc.
- The C character type routines defined in ctypes.h are useful for recognising characters. In particular, isblank. You should use at least one of the ctypes.h calls when parsing the dmake file.
- strsep should be used to separate the individual parts of a line of text, such as separating the file names in the rule header. This task is called tokenisation – each file name is delimited by white space (space or tab) and is a separate token. If you prefer, you may use other similar system library calls such as strtok, strchr, etc.

Advanced Features:

- malloc creates data structures on the heap. realloc is useful for dynamically resizing arrays of data or pointers, if you wish to dynamically allocate the memory. You should also free the data structures when they are no longer needed.
- strdup is useful for creating copies in the heap of strings that may be in local or global arrays. Strings created with strdup can later be freed using free.

Marking

The auto-marker will test your submission for this stage with the test files provided to you and with additional hidden test files. Some tests will be performed using the `-f` option and some will be performed by using an input file with the name `Dmakefile`. Therefore, your program must support both modes of operation. You may assume that the hidden test files have similar features as the test files that are provided to you – for example, instances of comments, of rules with and without commands, and both small and large files, and errors. While there are additional features that are introduced in later stages, your program will never be auto-marked using the `-z1` option with those additional features. This means that, in later stages, you can add those features to the `-z1` debug output (if you wish) without being concerned about failing the stage 1 tests because those features will not be tested with the `-z1` option.

When your `dmake` program is executed with the `-z1` option, it should print out the `-z1` debugging output and then terminate. It should not continue with stage 2 or other additional processing.

A **detailed marking guide** is provided to you when you download the stage. Please note the following:

- Most marks are awarded for correctly passing the test cases that are supplied to you.
- A significant number of marks are awarded for correctly passing the hidden test cases.
- Some marks are awarded for meeting specific programming requirements. In particular, you are expected to use relevant system calls and library routines to assist with the parsing of the input file. Also, for maximum marks, you are expected to dynamically allocate memory to store the information that you read from the input file, and the amount of memory used will vary depending on the size of the input file[3].

[2 marks] Stage 2: Determining the rules that will fire

The purpose of dmake is to execute the rules that need to be executed in order to bring the target files up to date with respect to the existing input files. In this stage, you will implement the code to decide which rules need to be executed, but without actually executing the rules.

Each rule header specifies one or more target files and zero or more dependencies. The meaning is the same as in `make`: If the target file does not exist, or if it is not more recent than all of the dependency files[4], then the rule should fire. When the rule fires, the commands in the rule would normally be executed – but in stage 2 the commands are not executed. After a rule fires we assume that all the targets have been updated. Therefore, any other rule that has one or more of the targets as a dependency will need to fire.

---

[3] This requirement does not mean that all the data structures have to be in the heap, only that there are some data structures in the heap and the amount of such data varies from one input file to another.

[4] The wording "not more recent than all of the dependency files" may seem difficult. The point is that if the target happens to be exactly the same age as one or more of the dependency files then we cannot be sure that the target was built from the latest version of the dependency file, so in this case the rule should fire. Also, if the target is older than any of the dependency files then the rule should fire.

```
# echo: Build the echo program
echo: echo.o
  gcc -o echo echo.o
echo.o: echo.c
  gcc -c echo.c
```

Figure 1 A simple Dmakefile

Process the rules from last to first

As a simple example, consider the dmake file in figure 1. This file contains two rules. The first rule builds the program `echo` from the object file `echo.o`, while the second rule builds `echo.o` from the source file `echo.c`. If `echo.c` is no older than `echo.o`, then the second rule should fire to update `echo.o`. Subsequently, the first rule should fire to update the executable program `echo`. In this particular example, the second rule must be considered for execution before the first rule, because the first rule depends on a target `echo.o` of the second rule. We say that the first rule is a parent of the second rule, and the second rule is a child of the first rule. In dmake, parent rules come before their child rules. This simplifies processing of the rules.

Child rules need to be considered for execution before their parents. Since child rules always come after their parents in dmake, you should process the rules in reverse order – i.e. start by considering the last rule and work back towards the first rule. As you consider each rule in turn, you will decide whether it needs to be executed or not.

Determine whether a rule will fire

To decide whether to fire a file you need to consider the timestamps of the dependency files and the timestamps of the target files. The simplest case is when the dependencies and targets all exist as disk files – dmake will fire the rule if any target is older than any dependency, or is the same age as any dependency. To put it simply: if the target is not newer than the dependency then we need to update the target.

You can find out which dependencies and targets exist, and their modification times, by using one of the variants of the system `stat` call (see `stat`(2)). `Stat` is a system call that obtains status information about files. It returns a lot of potentially useful information, but for our purposes the most important information is the modification time, which is the last time that the file was changed. `Stat` can also tell us if a file does not exist.

If the target of a rule does not exist then the rule should fire in order to create the target. For example, in figure 1, if `echo.o` does not exist then the second rule should fire in order to create `echo.o`. Similarly, if `echo` does not exist then the first rule should fire to create `echo`. These and other use cases are presented in the next section.

In stage 2 we are not actually executing rules, so the target files will not actually be updated. In this case we pretend that firing a rule has updated the target. What this means is that you need to know the names of all the targets of all the rules that have already fired. You can assume that each of these targets is newer than any file on disk, and if one of these targets appears as the dependency of another rule than that rule must fire.

If a dependency of the current rule does not exist and is not a target of a child rule, then dmake cannot proceed with the current rule and reports an error. In figure 1, this would happen if `echo.c` did not exist because there is no way for dmake to create `echo.c`.

Use case	`echo.c`	`echo.o`	`echo`	Fire rules
1	Sep 9, 10:30	Sep 9, 10:35	Sep 9, 10:40	None – up to date
2	Sep 9, 10:30	Sep 9, 10:35	Not exist	1 – partial build
3	Sep 9, 10:30	Sep 9, 10:40	Sep 9, 10:35	1 – partial build
4	Sep 9, 10:30	Not exist	Not exist	2, 1 – full build
5	Sep 9, 10:30	Sep 9, 10:20	Sep 9, 10:25	2, 1 – full build
6	Sep 9, 10:30	Sep 9, 10:25	Sep 9, 10:20	2, 1 – full build
7	Sep 9, 10:30	Not exist	Sep 9, 10:40	2, 1 – full build
8	Not exist	Any	Any	None - Error

Figure 2 Use cases for figure 1 Dmakefile

In summary, to determine whether a rule will fire or not, examine the dependencies and the targets. The rule must fire if any target is not more recent than any dependency, or if any target does not exist, or if any dependency is the target of a child rule that has fired.

Use cases

Consider the dmake file in figure 1. Figure 2 shows the existence and timestamp of each file in various use cases. For simplicity, times are shown with clear differences in the minutes. Assume that the current time is Sep 9, 11:00.

Case 1 is where the program has been compiled since the last time that `echo.c` was edited. `echo.o` is more recent than `echo.c`, and `echo` is more recent than `echo.o`. No rules fire because everything is up to date.

Cases 2 and 3 are where `echo.o` is more recent than `echo.c` so rule 2 does not fire. However, in case 2 `echo` does not exist while in case 3 `echo` is out of date compared to `echo.o`. In both cases, rule 1 fires to rebuild `echo`.

In cases 4 through 7, both rules fire. Rule 2 fires either because `echo.o` is out of date with respect to `echo.c` (cases 5 and 6), or because `echo.o` does not exist (cases 4 and 7). Once rule 2 has fired, `echo.o` is newly updated, so rule 1 must fire. In case 4, rule 1 must fire because `echo` does not exist. In cases 5, 6 and 7, `echo` is out of date with respect to the new time of `echo.o` which is the current time Sep 9, 11:00.

In case 8, rule 2 detects an error because `echo.c` does not exist. As a result, no rules fire irrespective of the times of the other files.

Supplied test cases

The following information will be useful for investigating the supplied test cases.

- A README.txt file is provided in each test case directory. It briefly describes the test case.
- The expected standard output, error output, and the command that is used to run the test are provided in the files test*n*.out, test*n*.err and test*n*.cmd.
- The following command shows the time stamps of files in the current directory to 1ns resolution.
  ```
  $ ls --full-time
  ```

Requirements for stage 2

The primary requirement for stage 2 is to print, as debug output, the rule numbers of the rules that would be executed. The order of listing is important because the rules must be fired in a particular sequence. This is debug output only and the commands are not actually executed in this stage. Use the same output format as the examples provided.

Specific requirements are as follows.

- Implement the option `-z2` which tells `dmake` not to execute commands but only to determine which rules would fire. If a rule fires, then the targets of that rule are all treated as having been updated to the current time, irrespective of whether the corresponding disk file exists or not.
- Using the option `-z2`, produce debugging output the same as the samples provided.

Useful system and library calls

- `stat`(2) system call provides essential information about files. You can easily obtain the modification time to 1 second resolution. Actually, the modification time is available to 1 ns resolution, but that will not be required for the test cases in stage 2.
- `time`(2) system call gets the current time in seconds[5].
- `strcmp`(3) is useful for matching names of dependencies and targets.

Marking

The automarker will compile and test your dmake program using test cases that have been supplied to you for stage 2, and additional hidden test cases. The hidden test cases have similar characteristics to the test cases that are supplied to you. The test cases include dmake projects in various states of being up to date. Remember that each test case includes the expected output and the command that is used to run the test case.

All testing for stage 2 will involve the command line option `-z2` or `-z 2`.

A **detailed marking guide** is provided to you when you download the stage. Please note the following:

- A significant number of marks are awarded for correctly passing the test cases that are supplied to you.
- Most marks are awarded for correctly passing the hidden test cases.
- Some marks are awarded for meeting specific programming requirements. In particular, you are expected to use the system call stat (in one of its forms).

[2 marks] Stage 3: Executing commands

The primary task of `dmake` is to execute the commands that will create or update the targets. In this stage you will implement this feature.

Echoing commands

In normal behaviour, dmake first prints the command that it is about to execute. The command is printed to stdout. Printing the command is helpful to the user if the command happens to produce some output. After printing the command, it is necessary to flush the output (see `fflush`(3)) so that the displayed command appears on the screen before the output from execution of the command. This is necessary because stdout is buffered. If you don't use `fflush`, the output may appear correct on the screen but will not be correct when it is redirected to a file for testing.

[5] If you wish to work to 1ns resolution, clock_gettime(2) would be useful.

How to execute commands

To execute the commands, `dmake` must `fork` and `exec` a process which will run the command. Your program must use the `fork` system call and one of the `exec` family of calls for this purpose.

The `dmake` program must `wait` for completion of each command, and capture the exit status. It must check for non-zero (error) exit status which would indicate that the command encountered an error. If an error is detected, `dmake` must report the error, and then itself terminate with the same error code. Normally, `Dmake` will not attempt to execute any other rules, or even to complete the commands of the current rule, once an error is encountered.

If the command is terminated by a signal, dmake reports the signal number in an error message and then itself terminates with error code 1.

The simplest way to actually execute each command is to invoke the shell `/bin/sh` using the `-c` option to pass it a command[6]. The entire command is passed as a single parameter to `/bin/sh`. See the Unix manual pages for information on executing commands with `/bin/sh`.

For example, suppose that the command in the dmake file is

```
gcc -c echo.c
```

then the parameter list passed to exec would consist of three strings as follows.

```
/bin/sh
-c
gcc -c echo.c
```

As is always the case when calling exec, the first parameter is the name of the command that is being executed. In this case it is "`/bin/sh`".

The second parameter is the option "`-c`" and the third parameter is the entire command "`gcc -c echo.c`".

Updated timestamp of rule targets

After executing the commands within a rule, `dmake` takes note of the updated the timestamp of each target, using that timestamp to decide whether or not to execute subsequent rules. For targets which do not exist as disk files after execution of the rule (called *phony* targets), the updated timestamp is the current time. For targets that exist as disk files after execution of the commands, the updated timestamp is the time of file modification – usually, this will be very close to the current time.

Modified behaviour of dmake command execution

In dmake, as in make, the default execution of commands can be modified by special characters at the front of the command. Each special character causes a particular modified behaviour. The special characters can be combined in any order, but they must all appear before the command name itself. The modifiers may optionally be separated from the command name with white space, and there may be white space between the modifiers. The modifiers are removed from the command line before it is passed to `/bin/sh` to be executed.

[6] GNU make uses the shell to execute the commands.

Modifier	Modified behaviour	Example
`@`	dmake does not print the command that will be executed.	`@echo Create directory`
`–`	dmake does not terminate if the command exits with an error code or signal	`- rm *.o`
`=`	dmake does not warn the user if the command exits with an error code or signal	`=mkdir build`
`*`	dmake suppresses standard output of the command	`* grep ok output.txt`
`&`	dmake suppresses standard error of the command	`&gcc –c buggy.c`

Figure 3 Command behaviour modifiers in dmake

Figure 3 shows the command behaviour modifiers for dmake. The '@' and '–' modifiers and standard features in GNU make. The other modifiers are additional features for dmake.

The '@' modifier is particularly useful with the system `echo` command to print information to the user, as shown in figure 3. Without the '@' modifier, dmake will print the `echo` command including its parameter text and then the `echo` command itself will print the parameter text. The '@' modifier produces cleaner output by not displaying the command.

The '–' modifier is useful when a dmake session should succeed even though a particular command may encounter an error. When the '–' modifier is used, dmake reports to the user that the command exited with error, but dmake does not itself terminate. A typical example, shown in figure 3, is removing intermediate files – if the files do not exist then `rm` will fail, but the dmake session can still proceed because the files do not exist.

The '=' modifier is also used when a command may fail but the dmake session can still succeed. Like the '–' modifier, dmake does not terminate because of the command's error exit. The difference is that dmake does not even tell the user that the command exited with an error code. For example, the `mkdir` command in figure 3 will fail if the `build` directory already exists. Using the '=' modifier means that dmake does not terminate and does not even warn the user that `mkdir` exited with error. However, `mkdir` still prints its own error message.

The '*' modifier is useful for suppressing unwanted output from commands. For example, `grep` prints the lines that match the pattern and exits with code 0, but if no lines match then it exits with an error code. Using the '*' modifier in figure 3 suppresses the printed lines that match the input pattern, so `grep` can be used simply to raise an error if the pattern is not matched without showing the successful matches that occur when there is no error.

The '&' modifier suppresses unwanted error output from commands. For example, `gcc` prints error messages and exits with an error code if compilation fails. Using the '&' modifier in figure 3 suppresses the error output, hiding the compilation errors from the user, but dmake can still respond to the error exit code.

The modifiers can be combined. For example, combining '=', '*' and '&' has the effect of ignoring all error conditions and suppressing all output. Thus, the following `mkdir` command will fail silently and dmake will continue if the directory `build` already exists[7].

```
    = *& mkdir build
```

Implementation of the modified behaviours

The '@', '−' and '=' modifiers are easily implemented inside dmake because they directly modify dmake's behaviour. '@' modifies dmake's display of the command that is about to be executed, while '−' and '=' modify dmake's response to an error exit.

The '*' and '&' modifiers, however, require changing where standard output or standard error are directed while the command is executing. This means that you have to redirect stdout or stderr before executing the command. You achieve this by closing the existing stdout or stderr, and opening the null device `/dev/null` as either stdout or stderr. In Unix, the null device `/dev/null` discards all data that is written to it. Since you want to discard stdout or stderr for the command only, but not for dmake itself, you need to be sure to make the changes to stdout or stderr within the child process – after the `fork` but before the `exec` system call. Changes made in the child process will not affect stdout or stderr in dmake itself. Implementation should only require a few lines of C code. The system library function `freopen` is one option for implementing this feature. Another options is `fclose` followed by `open`.

Requirements for stage 3

When `dmake` is run, it will perform the rule traversal as in stage 2 and actually execute the commands in the rules that fire. The commands within a rule are executed in the sequence that they appear in the `Dmakefile`; they are to be executed when the `dmake` logic from stage 2 determines that the rule fires.

Your dmake program should evaluate the rules in the dmake file in each test case, execute those rules that fire, and produce output that matches the supplied output except for minor differences in white space. Please refer to the test case sample standard output and error output for examples of the syntax of error messages produced by dmake when a command exits in error, with and without the '−' command modifier. The README.txt file in each test case describes the expected behaviour of the test case, and the test*n*.out and test*n*.err files show the output of each test run.

Your dmake program should implement the command line modifiers listed above.

Your dmake program must use `fork` and one of the `exec` family of system calls to execute the commands. It must also use one of the `wait` family of system calls.

Your dmake program must implement the '*' and '&' modifiers itself, not by using I/O redirection capabilities of the shell `/bin/sh`.

Testing your dmake program in stage 3

Testing your program in stage 3 is more difficult than in previous stages because successfully executing dmake will typically change the files in the test case folder. To make testing easier, we recommend that you download the `stage3.tar` file into your project directory and then extract

[7] Unfortunately, `mkdir` will also fail silently for other reasons that prevent the directory from existing, such as access permissions on the working directory. The command modifiers provide very coarse control. Many system commands provide command-line options that are preferable to the dmake modifiers.

the `stage3.tar` file inside your project directory. In the following example, we assume that your project directory is called `assignment3`, and that this is where your dmake program source code resides.

```
$ cd assignment3
$ /home/units/comp202/lab -g 3.3
$ tar xvf stage3.tar .
```

Do not rename either the resulting `stage3` directory, or any of the files or directories within it. This means that you will have a `stage3` directory with subdirectories `test1`, `test2`, etc., all within your project directory.

When you want to run a test, change to the specific test directory and execute your dmake program. We assume that your dmake program is the in project directory, two levels up from the test directory, so the command is as follows (possibly with the `-f` option, if required).

```
$ ../../dmake
```

After executing a test, return to your project directory and then you can restore the test cases as follows.

1. Remove the test cases.

   ```
   $ rm -rf stage3
   ```

2. Extract the test cases from the original tar file again.

   ```
   $ tar xf stage3.tar .
   ```

You may wish to save these commands into a shell script or make file to reduce typing and the risk of errors.

Useful system and library calls

- `fork`(2) creates a new process, as discussed in lectures.
- `exec`(2) (or family member) starts executing a new program in the current process.
- `waitpid`(2) (or family member) waits for termination of the executed process.
- `freopen`(3) closes the current file descriptor of a `FILE *` (which can be `stdin` or `stdout`) and reopens it with a new file. You can also use `fclose`(3) together with `open`(2) to achieve the same effect.

Marking

The automarker will compile and test your dmake program using test cases that have been supplied to you for stage 3, and additional hidden test cases. The hidden test cases have similar characteristics to the test cases that are supplied to you. The test cases include dmake projects in various states of being up to date and with various command behaviour modifiers. Remember that each test case includes the expected output and the command that is used to run the test case.

There is no specific command line option for stage 3.

A **detailed marking guide** is provided to you when you download the stage. Please note the following:

- A significant number of marks are awarded for correctly passing the test cases that are supplied to you.

- Most marks are awarded for correctly passing the hidden test cases.
- Some marks are awarded for meeting specific programming requirements – the use of required system or library calls: fork, exec family, wait family, open or fopen family for IO redirection.
- You can achieve progress without implementing the command modifiers '`*`' and '`&`'.

[2 marks] Stage 4: The difference

One of the annoying features of `make` is that small edits can cause the entire project to be rebuilt. For example, if you edit a comment in a C source file, then that file's modification time is updated so the file is recompiled. The resulting object file is the same (because you did not change the code) but it has a new timestamp so everything that depends on the recompiled object file is also rebuilt. Of course, this behaviour does not produce errors – it merely wastes some time rebuilding parts of the project unnecessarily. For small projects, the impact is not noticeable, but for large projects (such as an OS kernel) it may take a long time to rebuild everything that is potentially affected by the edited file.

The *difference* in `dmake` is that it is intended to detect trivial changes and suppress unnecessary execution of other rules. This means comparing each rebuilt file with the corresponding old built file, looking for differences and deciding whether to fire other rules based not only on the current timestamp but also on the file content differences.

The essential requirements of this stage are as follows:

- If the target of a rule is unchanged by firing the rule, then other rules should not be fired simply because of the new timestamp of the target.
- Once dmake has determined that firing a rule has made no difference to the target, it should remember that fact so that it does not fire that rule again until one of the dependencies of the rule has changed.

Design

We present a suitable design for this feature. You are welcome to explore alternative design ideas if you wish.

Detecting differences

The first core task is to determine whether executing a rule has rebuilt the targets with the same content that they had before the rule was executed. To do this, we need to compare the contents of the target file before and after execution of the rule.

For example, consider the dmake file in figure 1 together with use case 5 in figure 2. Echo.c is more recent than echo.o so rule 2 will fire to recompile echo.c to echo.o. However, suppose that it turns out that the rebuilt echo.o is identical to the previous echo.o. In that case, rebuilding echo would yield the same result also, and it is actually unnecessary to fire rule 1. So we need a way to determine whether the rebuilt echo.o is the same as the previous echo.o or not.

A simple approach is to save the old echo.o file before executing rule 2, then compare the saved file with the new echo.o file that results from executing rule 2. The easiest way to save the echo.o file is to rename it, giving it another name that (probably) won't be used for any other purpose. In Unix, file names that begin with a period '.' are commonly used for hidden storage because the `ls` command does not show them unless you use a specific command-line option. We will save echo.o as the file **`.~echo.o`** – the file name is copied from the original but has the two characters period and question mark (`.~`) inserted in front of it. By this method, any existing target file can be saved – simply rename it with `.~` inserted at the front of the file name.

> Before executing a rule, save each of the target files (if they exist) by using the system call `rename(2)` to change their name, adding `.~` in front of the file name.

After executing the rule, we compare the contents of the saved file **`.~echo.o`** with the newly built file `echo.o`. This means that the two files have to be opened, read and compared. You can open them using `fopen(3)` and read them character by character with `getc(3)`, or you can open them with `open(2)` and read them in chunks using `read(2)`. Either way, you say that the files are different if there is one or more bytes of data that differ, or if the two files are different lengths.

> Using `fopen(3)` and `getc(3)` is the easiest way to write the code. In a loop, you read a byte from each file, check whether you reached end of file, then compare the bytes.
>
> Using `open(2)` and `read(2)` is more complicated but operates at the system call level so there is a small mark component for using `open` and `read` with blocks of at least 100 characters in each `read` call.
>
> There is also a small mark for using the `stat`(2) system call to detect a difference in the sizes of the files so that you only actually read them if they are the same length.

If the rebuilt file is the same

If the rebuilt file is the same, then dmake may be able to avoid executing other rules. For example, with the dmake file in figure 1 and use case 5 of figure 2, if rebuilding `echo.o` makes no change to `echo.o` then there is no need to rebuild `echo`. The following describes how you can implement this optimisation of dmake's behaviour.

Firstly, you don't want to keep two identical copies of `echo.o`. Rename `.~echo.o` back as `echo.o` – this will remove the rebuilt `echo.o` file. Since dmake now has an `echo.o` file which has not been updated, rule 1 will not fire in use case 5 because `echo` is more recent than the restored `echo.o`.

However, we have deleted the rebuilt `echo.o` so there is nothing in the file system to tell us that `echo.c` should not be recompiled the next time that dmake is run – remember that `echo.c` is more recent than the restored `echo.o`! To solve that problem, we create an empty *memo* file called **`.@echo.o`** to remember the timestamp of the latest time that `echo.o` was rebuilt. The only information that is important in this file is its modification time, so the file can be completely empty. You can create an empty file by opening/creating a file and then immediately closing it, and the timestamp of the file will be the time when you created it. The simplest way to create a file is to use `fopen`(3), but the Unix way is to use `open`(2) with flags indicating to create the file.

If the rebuilt file is different

If the rebuilt file is different then dmake should discard the saved file (e.g. **`.~echo.o`**) and then continue as normal. The system call `unlink`(2) will be useful for removing the saved file.

If there exists a memo file corresponding to the rebuilt file (e.g. **`.@echo.o`**) then the memo file is also removed because it is no longer relevant.

If the target file does not exist

If the target file does not exist before running the rule then dmake performs the same as in stage 3. If the target still does not exist after running the rule, then dmake continues to perform the same as in stage 3.

The effect on other rules

When you are deciding whether or not to fire another rule, you need to consider the timestamp of the `echo.o` file and also the timestamp of the corresponding memo file `.@echo.o` if a memo file exists.

If the file is a dependency of the rule, then the timestamp of the file itself is what you use for comparison. For example, when you are deciding whether to fire rule 1, compare the timestamp of `echo.o` with the timestamp of the program `echo`.

If the file is a target of the rule, then you need to check the timestamp of the memo file as well as the timestamp of the target file itself. If the memo is older than the file itself, then the memo is no longer useful because the file itself has been rebuilt more recently, so the memo should be deleted and the timestamp of the file should be used. However, if the memo is newer than the file, then use the memo timestamp to decide whether to fire the rule. For example, suppose you run dmake a second time and rule 2 is being considered. Because the memo file `.@echo.o` is more recent than `echo.c`, rule 2 will not fire.

The test cases provided to you include examples with memo files, both memo files that suppress the execution of rules and memo files that are no longer relevant.

Requirements for stage 4

When dmake is run with the `-z4` option, it will execute rules as in stage 3, but with the differences capability described in this section. In summary:

- If executing a rule rebuilds a target with no change, then dmake will restore the original target file and will create a memo file to prevent the same target triggering the same rule each time dmake is run.
- When considering dependencies, dmake uses the timestamp of the file itself.
- When considering targets, dmake examines the timestamp of the memo file, if one exists.
- Memo files that are out of date are deleted when they are discovered as described above.
- Use unlink to delete files that are not needed.
- Use rename to save files.

For top marks, the following optional requirements are checked by the automarker

- Use `stat` during file compare to avoid reading files of unequal lengths.
- Use `open` and `read` (with suitably sized read blocks) rather than `fopen` and `getc` to compare files.
- There is one supplied test case that has more than one target in a rule.

Useful system and library calls

- `stat`(2) to examine the time of files, including memo files, and the examine file size.
- `rename`(2) to rename a file.
- `unlink`(2) to remove a file.
- `open`(2) or `creat`(2) to create a file.

- `close`(2) to close a file opened or created with open/creat.
- `read`(2) to read a block from a file
- `fopen`(3) and `getc`(3) as an easiuer alternative to `open`, `creat`, `close`, and `read`.

Testing your dmake program in stage 4

You can use similar testing as in stage 3. However, you should know that the automarker monitors your system calls and can identify whether you have used them appropriately. The automarker will tell you which test cases fail, and which test cases fail because the directory is incorrect at the end of the test. Usually, the directory is incorrect when your dmake program does not make the correct decisions about which rules to fire, but it will also be incorrect if you do not manipulate the memo files correctly.

You can check the directory yourself. Each test example is provided with a `test`*`n`*`.ls` file. This file shows the files that are present in the `test`*`n`* directory after successful execution of `dmake -z4`. The files are listed in reverse order of modification time – the most recently modified file is listed first. The automarker checks that execution of your dmake program leaves the directory in this same condition – any additional files, missing files, or files out of sequence are considered to be an error. Also, any files that are not the same size (expressed in file system disk blocks).

After a test run of your program, you can produce the corresponding directory listing by running the following system command inside your `test`*`n`* directory. Note, the command includes the digit '`1`' as an option – it is not the letter '`l`'. The '`1`' option lists the files in a single column.

```
$ ls -A1ts
```

The automarker will report an error to you if the directories do not match for a supplied test case. You can also run the above command, saving the output to a text file and compare it with the supplied listing. For example, suppose that you are in the `stage4` directory which contains the supplied test cases as subdirectories; you can check the directory listing for `test1` using the following commands.

```
$ ls -A1ts test1 > test1.myls
$ diff test1.ls test1.myls
```

In fact, a single command can be used. The following command tells `diff` to compare the file `test1.ls` with input arriving on standard input (which is a pipe). It is a common Unix convention that a single hyphen as a file name parameter represents reading from standard input.

```
$ ls -A1ts test1 | diff test1.ls -
```

Warning: `diff` tells you the differences between the directory listings *as text*. If a file is in a different place in the listing, then diff will highlight that fact, and much of the time it will tell you which file is incorrect. However, sometimes the file that it highlights as the difference may not actually be the problem. For example, consider the following two simple directory listings.

```
echo            echo.o
echo.o          echo
echo.c          echo.c
```

In this example, the files echo and echo.o are interchanged in the second listing. When `diff` compares these listings, it may report either echo.o or echo as the misplaced file. When two adjacent files are interchanged, as in this example, there is no way for diff to know which file name is actually out of place. Diff shows you that the two files are out of sequence with respect to each other. It is up to you to work out whether the underlying problem is with echo or with echo.o.

Your final marks for lab 3

Your final mark for lab 3 is the sum of the individual stage achievement marks (which are scaled up to each be marked out of 3.0), plus the progress marks that you earned throughout the lab.

The lab command can show you the total mark for lab 3:

```
$ lab -m 3
```

And it can show you the total mark for all labs:

```
$lab -m
```