
The University of Queensland
School of Information Technology and Electrical Engineering
Semester One, 2009

COMP2303 / COMP7306 – Assignment 3

Due: 11pm Saturday May 16, 2009

Marks: 50

Weighting: 25% of your overall assignment mark (COMP2303)

Introduction

The goal of this assignment is to give you practice at C programming and the use of a small number of system calls involved with getting information from the file system and executing processes. You are to create a simplified version of the `make(1)` program called `2303make`. Like `make(1)`, your `2303make` program should be able to build a target file (typically an executable) from some source files and only rebuild when the target is out of date with respect to the source files (dependencies). You should make sure you understand how `make(1)` works before you proceed. The assignment will also test your ability to code to a programming style guide.

This is an **individual assignment**. You should feel free to discuss aspects of C programming and the assignment specification with your fellow students, but you shouldn't actively help (or seek help from) other students with the actual coding of your assignment solution. It is cheating to look at another student's code and it is cheating to allow your code to be seen or shared in printed or electronic form. You should note that all submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct proceedings will be initiated against you. A likely penalty for a first offence would be a mark of 0 for the assignment. Don't risk it! If you're having trouble, seek help from a member of the teaching staff – don't be tempted to copy another student's code. You should read and understand the statements on student misconduct in the course profile and on the school website:

http://www.itee.uq.edu.au/about_ITEE/policies/student-misconduct.html

Assignment Specification

Command Line Arguments

Your program is to accept command line arguments as follows:

```
2303make [-f filename] [-depend | target-name]
```

where `[]` indicates optionality, `|` indicates a choice and italics indicate that arguments are to be replaced as appropriate. The argument pair `-f filename` is optional, and if specified, is the pathname of the rule file which contains the details of how to build the target file (i.e. the equivalent of a `Makefile` to `make(1)`). If `-f` is present on the command line (can only be the first argument) then there must be at least one argument following it and that argument (whatever it is) is taken as the rule file name. If the filename is not supplied, then your program should assume the name of the rule file is `2303makefile` and that this file is located in the current directory. The remaining (last) argument (if present) must be either `-depend` or a `target-name`. The meaning of `-depend` is described below (advanced functionality). The `target-name`, if given, is the primary target to be rebuilt (typically an executable, but doesn't have to be). If no `target-name` is given, the primary target is the first named target in the file. The `target-name`, if present, will be the last argument and can have any value other than `-depend`. Your program

should report an error and exit if the supplied command line arguments do not match those expected. (See error reporting below.)

If the rule file does not exist, then the program should report an error and exit. (See below.)

Rule File Syntax

Rule files (2303makefiles) are ASCII text files, consisting of lines of any length terminated by standard UNIX newline characters (\n or ASCII code 10). Lines can have one of the following forms (ignoring the newline character at the end of each line). Note that you will be supplied with code which reads in a rule file to a data structure so you need not be too concerned about the syntactical details below.

1. **Blank lines** – i.e. lines containing nothing or any number of white space characters (as determined by the `isspace(3c)` function). Such lines are ignored when processing the file.
2. **Comment lines** – i.e. lines which start with a # character or start with any number of white space characters followed by a # character. Any characters may occur after the # character. Such lines are to be ignored when processing the file. (Note, comments can only appear on lines by themselves – not after text on other lines.)
3. **Definition lines** – i.e. lines which have the format
name = [*string*]
The line must begin with a non-space character and contain a single equals sign character. The name is all of the characters up to the first whitespace character or equals sign. The “value” of the given name is the string given after the equals sign (if any). All whitespace characters immediately either side of the equals sign and at the end of the line are ignored. (Note that this means it is not possible to define a name as being equal to a space or some combination of whitespace characters.) It is permissible to have nothing on the right hand side of the equals sign. Definition lines may not have any colon characters. A name can only be defined once.
4. **Dependency lines** – i.e. lines which have the format
filename : [*pathname*]
Dependency lines indicate that the given *filename* (target) is dependent on the given file *pathname* (which could be an absolute or relative pathname). In other words, if the target filename doesn't exist or has an earlier creation date than one of its dependencies, then the target should be rebuilt. The line must begin with a non-space character and contain a single colon character. The filename is all characters up to the first whitespace character or colon. The pathname is all characters after the colon, if any (with leading and trailing whitespace characters removed). All whitespace characters immediately either side of the colon and at the end of the line are ignored. Dependency lines may not have any equals sign characters. Dependency lines can be followed by 0 or more rule lines that list commands to execute to create the target filename from its dependencies. Dependency relationships can be present multiple times in the file if desired.
5. **Rule lines** – i.e. lines which can only immediately follow a dependency line or another rule line and specify a command to be executed (possibly as part of a sequence of commands) to rebuild the associated target. A rule line must start with a single tab character which is followed by a command that must be executed (possibly as part of a sequence of rule commands). All characters after the tab make up the command – with whitespace characters separating the arguments. For each unique target name, there must be at least one rule. All rules for a target name are grouped together in the order given in the file – even if they are associated with different dependency lines.

The following is an example of a valid 2303makefile:

```
GCC=gcc -Wall -ansi -c
LINKER = gcc -Wall -ansi
wordsearch : wordsearch.o
    $(LINKER) -o $* wordsearch.o
    chmod 700 $*
    ls -al $*
wordsearch.o:wordsearch.c
wordsearch.o : wordsearch.h
wordsearch.o: /usr/include/stdio.h
    $(GCC) wordsearch.c
clean:
    rm wordsearch
    rm wordsearch.o
```

2303makefile Semantics – What Does 2303make do?

The 2303makefile is interpreted as follows. The primary target is either that named on the command line, or the first named target in the file. 2303make will try to build this primary target (if necessary). 2303make will examine the existence and/or modification times of all of the target's dependencies and their dependencies etc and rebuild the targets as necessary all the way up to the primary target.

Rebuilding a target involves executing all the rules associated with that target (in the order specified in the 2303makefile). Before execution, these rules will have substitution undertaken on them. The following substitutions happen:

1. `$*` is replaced by the name of the target for that rule
2. `$(NAME)` is replaced by the definition value for the given defined name. If the NAME is not defined, then no substitution will happen. Definitions are not further expanded – if a definition value has something that looks like `$(NAME)` or `$*`, it is not further expanded. Definition values may have whitespace, which means one argument in the rule may turn into several arguments (or no arguments if the definition value is empty).

You do not need to perform other substitutions, e.g. the argument `*` will not be substituted with the names of the files in the current directory. (This is something that shells do – your 2303make program does not have to do this.)

If there are rules associated with a target, but no dependencies (e.g. the target “clean” in the example above), then the rules will always be executed if that target is a primary target. Note that if there are no dependencies of a target, then the target need not be created as a file. However, all dependencies must be files (or created as files) – e.g. the target “clean” in the example above, can never be a dependency of another target unless one of the rules associated with the target creates a file called “clean”.

You can assume there are no dependency loops (e.g. A depends on B depends on A). In other words, your program can behave however you like, including crashing or going into an infinite loop if the 2303makefile contains dependency loops.

If a rule command ever fails (returns exit status other than 0) or if a target is not created after the execution of all the rules associated with the target or if the target still has a modification time earlier than any of its dependencies after the execution of all the relevant rules then an error will be reported (see below) and the build process will be halted. (Targets without dependencies need not exist after their rules are run.)

Other functionality

All commands executed by `2303make` should be printed to standard output just before execution. (Printing happens after substitution occurs.) `2303make` itself should print nothing else to standard output, but the standard output *and standard error* of the compiling and linking commands executed by `2303make` should also appear on standard output. If nothing is rebuilt, nothing should be printed and `2303make` should just exit with exit status 0.

When commands are executed, they should be run with the same environment as `2303make` has.

You may NOT use the `system(3c)` library function. You may not execute a shell program.

Error reporting

If `2303make` needs to exit because of an error condition it must print a suitable message to standard error and exit with a non-zero status code (see below). `2303make` itself should print nothing else to standard error. The standard error output from rule commands executed by `2303make` must be redirected to `2303make`'s standard output.

The errors to be reported are described below. All of the errors associated with the contents of the `2303makefiles` themselves (error code 1) will be checked for and reported in the code supplied to you.

Error	Message to be printed to standard error (\n is a newline character)	Exit status
Invalid lines in the <code>2303makefile</code> . (%d is replaced by the line number)	Error: Line %d: Invalid line\n	1
No targets specified in the <code>2303makefile</code>	Error: File must contain at least one target\n	1
Error reading the <code>2303makefile</code> . (This is after the file is open.)	Error encountered reading file\n	1
A target doesn't have at least one rule associated with it. (%s is replaced by the target name)	Error: Target %s does not have any rules\n	1
A name is defined more than once in the file. (%d is the line number of the second occurrence; %s is replaced by the definition name.)	Error: Line %d: Name %s is already defined\n	1
Usage error – if the command line arguments are not as expected. (Message is output on a single line.)	Error: Usage: 2303make [-depend] [-f file-name] [target-name]\n Error: Usage: 2303make [-f file-name] [-depend target-name]\n	2
Unable to open <code>2303makefile</code> (or the filename supplied on the command line). (%s is replaced by the name of the file.)	Error: Unable to open file %s\n	3
If a target is specified on the command line but is not listed in the <code>2303makefile</code> (%s is replaced by the name of the target)	Error: Target %s not in 2303makefile\n	4

Error	Message to be printed to standard error (\n is a newline character)	Exit status
Execution of a rule fails (either can't be executed or returns a non-zero exit status)	Error: Command failed - aborting\n	5
A target used as a dependency is not created or is still out of date after running all the rules associated with the target. (%s is replaced by the name of the target.)	Error: Target %s not updated\n	6
A specified dependency is unable to be found or its status obtained (and there is no target/rule to build it). (%s is replaced by the name of the dependency.)	Error: Unable to get status on %s\n	7

If the primary target is successfully created, your program should exit with status 0.

Dependency Generation Functionality (Advanced Functionality)

The functionality described in this section is the advanced functionality for this assignment.

If the `-depend` argument to `2303make` is specified, then the program should automatically generate a list of header file dependencies for all of the object files that make up the program. (In other words, for all `.o` files listed as targets, if there exists a corresponding `.c` file (i.e. name prefix is the same), then all the `.c` and `.h` file dependencies of that `.o` file should be added to the specified rule file (or `2303makefile` if no rule file name is specified on the command line.) This list should be appended to the specified rules file and be in the form of dependency lines as specified above. If such a list has previously been generated and appended to the file, it should be replaced rather than having a second version appended. The functionality of `2303make -depend` is similar to that of the `makedepend(1)` utility. The rest of the file should be unchanged – i.e. dependencies entered by the user (at the top of the file) should not be changed. Once the dependency list has been updated and the rule file saved, `2303make` should exit with exit status 0. (Nothing should be printed to either standard output or standard error **if the operation is successful. The exit status will always be 0. If the usage is incorrect or the rule file can not be opened for reading or writing then the program should exit with status 2 or 3 and print the message as described above. Errors 1,4,5,6 or 7 will not occur if the -depend option is given.**)

In order to know if a list of dependencies has been previously appended to the rules file, you should add a comment to the file, in a manner similar to `makedepend(1)`. The comment to be added is:

```
#TEXT BELOW THIS LINE HAS BEEN AUTOMATICALLY GENERATED
```

To generate the dependencies, you should capture and process the output of `gcc(1)` when run with the `-MM` option on the relevant `.c` files. (Do not print the commands being executed.) Note that `2303makefiles` do not support specifying multiple dependencies on one line, or line continuation with the backslash character, so you will need to parse the output of `gcc -MM` – you can't just append it directly. You must include all dependencies, including header files included by other header files etc. (`gcc -MM` will report all of these.) If `gcc -MM` fails (returns non-zero status) then no dependencies should be added for that target.

Specification Updates

It is possible that this specification contains errors or inconsistencies or missing information. It is possible that clarifications will be issued via the course website. Any such clarifications posted 5 days (120 hours) or more before the due date will form part of the assignment specification.

Style

You must follow **version 1.4** of the COMP2303/COMP7306 C programming style guide found at http://www.itee.uq.edu.au/~comp2303/resources/c_resources.html.

Hints

You may want to consider using the `stat(2)` system call (for determining file existence and modification times) and the `fork(2)`, `execvp(2)`, and `wait(3c)` (or `waitpid(3c)`) system calls for executing commands. The `argv` structures created in the code supplied to you are suitable for passing as the second argument to `execvp(2)` – but you will need to take care of any definition substitution first. You may also want to consider using `dup2(3c)` to redirect command standard error.

It is suggested that you use the following strategy in creating your program. You should test your program after every step and print out debugging information so that you are satisfied the program works to that point. Don't forget to disable the printing of the debugging information before you submit. If you are considering only implementing some of the required functionality then you should consult the marking scheme (below) and you may wish to follow a different strategy.

1. Write the code that checks the command line arguments and determines the name of the rules file. Create your initial Makefile (i.e. the Makefile to build your 2303make executable) and build and test your program.
2. Look at the supplied code for reading a 2303makefile and integrate it with your program. Create some test 2303makefiles. Use the supplied `print_makefile_data()` function to check that you're reading the files correctly.
3. Write code to identify the primary target filename (either from the 2303makefile or as given on the command line)
4. Write a function that determines the existence/modification time of a given file and use it to determine the status of the primary target file.
5. Write code to check the modification time of each of the dependencies of the primary target and determine whether the primary target should be built. (Don't worry about multiple levels of dependencies yet – just one level – and assume that the dependencies all exist and don't need to be rebuilt.)
6. Write code to execute the rule commands to rebuild a target (don't worry about definition substitution yet).
7. Add code to print the commands being executed
8. Add code to redirect command standard error to the top level program's standard output
9. Add checks for execution failure, failure to build the target, target remains too old
10. Add target substitution - `$(*)` syntax
11. Add definition substitution - `$(NAME)` syntax, ignoring whitespace
12. Add support for whitespace in definition substitutions
13. Write code to deal with multiple levels of dependencies, including intermediate dependencies that may not initially exist.

-
14. Add other error handling functions not yet implemented.
 15. Add other required functionality.
 16. Write code to add advanced functionality.

Submission & Version Control

You must use your COMP2303 Subversion repository for storing/submitting your source code for this assignment. Details will be provided on how you can access this. Your assignment must be placed in the `comp2303a3` directory within your repository. You must create a Makefile that will build your assignment. Do not add executables or object files to your repository. (The contents of any subdirectories within your repository `comp2303a3` directory will be ignored. You can create subdirectories if you wish, e.g. for test files, but none of those files will be considered when assessing your work.)

Note that a testing program will be supplied, but it will only test code from your repository.

Submissions (the contents of your `comp2303a3` directory in your repository) will be retrieved from your repository when submissions have closed (5 days after the due time, i.e. 11pm Thursday May 21). Your submission time will be taken as the commit time of the newest version within this directory at that time.

The due date for this assignment is 11pm Saturday May 16, 2009. The policy on “grace days” and exceptional circumstances (e.g. illness) is as outlined in the course profile (under “Late Submission”). Submissions completed before 11pm Friday May 15, 2009 will earn additional grace days. Note that no submissions can be made more than 120 hours past the deadline (i.e. after 11pm Thursday May 21, 2009) under any circumstances.

Code Compilation

The code retrieved from your repository will be compiled on `agave.students.itee.uq.edu.au` as follows:

1. The `comp2303a3` directory within your repository will be checked out to a testing directory on `agave`.
2. All subdirectories, executables and object files in your `comp2303a3` directory (if any) will be removed. (This should leave only `.c` files, `.h` files and a Makefile.)
3. For the purposes of counting the warnings generated, each `.c` file in your submission will be compiled with the command:

```
gcc -Wall -ansi -c filename.c
```

where *filename* is replaced as appropriate. (The object files created with this method will then be removed.)

4. The following command will be executed:

```
make 2303make
```

If any errors result from the command above (i.e. the `2303make` executable cannot be created) then you will receive 0 marks for functionality (see below). Any code without academic merit or any use of the `system()` library function or calls to a shell will be removed from your program before compilation is attempted (and if compilation fails, you will receive 0 marks for functionality).

Marks

Marks will be awarded for both functionality and style as described below.

Functionality (42 marks)

Provided your code compiles (see above), you will earn functionality marks based on the number of features your program correctly implements. Partial marks will be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. **If your program does not allow a feature to be tested then you will receive 0 marks for that feature**, even if you claim to have implemented it. For example, if your program can never run a command then it will not be possible to determine that your program correctly handles command failure. The markers will make no alterations to your code (other than to remove code as noted above), although minor corrections may be possible (see below). Marking will be automated – it is important that you follow the specification precisely, e.g., correctly spelling all error messages.

- A. Error handling - program handles invalid command lines (error 2) (2 marks)
- B. Error handling - program handles non existence of `2303makefile` (or the filename specified on the command line) (error code 3) (2 marks)
- C. Error handling - program handles invalid makefiles. (This functionality is provided to you, but marks are given here to ensure that you call it correctly and/or don't modify it incorrectly.) (error code 1) (2 marks)
- D. Error handling - program handles being given a target name on the command line which is not present in the `2303makefile` (error code 4) (2 marks)
- E. Error handling – program handles failed execution of a command (e.g. non-existent command) (error code 5) (2 marks)
- F. Program is able to execute a command (no substitution required) (2 marks)
- G. Error handling – program handles execution of a command that returns a non-zero code (must be able to execute a command first – this applies to many criteria below also) (error code 5) (2 marks)
- H. Error handling – program handles a target used as a dependency not being created by execution of the given rules (error code 6) (2 marks)
- I. Error handling – program handles a target used as a dependency being created but with an earlier modification time than at least one of its dependencies (error code 6) (2 marks)
- J. Error handling – program handles being unable to open or get the status time of a dependency (error code 7) (2 marks)
- K. Program is able to create a non-existent target from only one level of dependencies (target can come from the command line, or be the first target in the file) (2 marks)
- L. Program is able to recreate an existing out-of-date target from only one level of dependencies (2 marks)
- M. Program always creates a target that has no dependencies (2 marks)
- N. Program doesn't recreate targets that are not out of date (2 marks)
- O. Program is able to create/recreate a target from multiple levels of dependencies – both existent and non-existent (i.e. must be able to build intermediate targets) (2 marks)
- P. Program prints all commands that are executed and nothing else (other than command output) (2 marks)
- Q. Program redirects command standard error to its own standard output (and prints command standard output to standard output) (2 marks)
- R. Program handles definition substitution without whitespace (2 marks)
- S. Program handles definition substitution with no value or value containing whitespace (i.e. value turns into 0 arguments or more than one argument) (2 marks)
- T. **Advanced functionality** – initial creation of header dependencies correctly handled (2 marks)
- U. **Advanced functionality** – recreation of header dependencies handled (2 marks)

Style (8 marks)

Your style mark will be the minimum of:

$8 \times 0.9^{(\text{Number of style guide violations} + \text{number of compilation warnings})}$
and
your functionality mark.

The number of compilation warnings will be the total number of warning lines reported during the compilation process described above. The number of style guide violations refers to the number of violations of version 1.4 of the COMP2303/COMP7306 C Programming Style Guide. A maximum of 5 violations will be penalised for each broad guideline area. The broad guideline areas are Naming, Comments, Braces, Whitespace, Indentation, Line Length and Overall. For naming violations, the penalty will be one violation per offending name (not per use of the name) up to the maximum of five. You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final – it is the marker who has to understand your code. To satisfy layout related guidelines, you may wish to consider the `indent(1)` tool. Your style mark can never be more than your functionality mark – this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

Penalties

In cases where submitted code does not compile due to a minor error (e.g. failure to submit a required .h file), the course coordinator may make minor source code corrections and mark the “corrected” code. If this occurs, a penalty between -5 and -15 marks will be applied (at the discretion of the course coordinator) depending on the amount of code correction required. If changes are only required to one line of code, the penalty will be -5.

Late Penalties

Late penalties will apply as outlined in the course profile if no grace days remain at the time of submission.