UMEÅ UNIVERSITY
Department of Computing Science
Assignment 2 report

October 1, 2021

5DV088 - Assignment 2, v1.0

C programming and Unix, Autumn 2021, 7.5 Credits

Make implementation

Name	Elias Olofsson
user@cs.umu.se	`tfy17eon@cs.umu.se`

Teacher
Mikael Rännar

Graders
Jonas Ernerstedt, Elias Häreskog, Oscar Kamf

Contents

1 Introduction

In this lab, we have re-created a bare-bones version of the ubiquitous GNU make utility, used as an easy and powerful way to keep track of which parts of a program needs to be recompiled, and does it automatically for you. Since the `make` utility already exists, here we only aim to complete an exercise in replicating some of the functionality seen in the original `make`, training our ability to write code in C.

2 Compilation and execution

The final program `mmake.c` I have written can be compiled in a simple way using the original `make` utility and the included `Makefile`. This can simply be achieved by navigating to the appropriate directory and executing the command `make`, as seen in Fig.1 below.

```
elias@xps13:~/Documents/C-Unix/ou2$ make
gcc -g -std=gnu11  -Wall -Wextra -Wpedantic -Wmissing-declarations
-Wold-style-definition -Werror -Wmissing-prototypes -c mmake.c
gcc -g -std=gnu11  -Wall -Wextra -Wpedantic -Wmissing-declarations
-Wold-style-definition -Werror -Wmissing-prototypes -c parser.c
gcc  -o mmake mmake.o parser.o
```

Figure 1 – Compilation of `mmake.c`, using GNU make and the included `Makefile`.

Now the program should be compiled and linked, such that an executable `mmake` is ready at our disposal. It can simply be executed as seen in Fig.2 below.

```
elias@xps13:~/Documents/C-Unix/ou2$ ./mmake
mmakefile: No such file or directory
```

Figure 2 – Execution of the program `mmake` without arguments. No appropriate makefile was detected in the current directory.

As the error message alluded to, there was no valid makefile detected in the current working directory. However, the program is successfully compiled and seems to be working, which means that we can move on to the next step.

3 User guide

3.1 Getting started

As we talked about in the introduction, this program reads in a makefile and executes commands depending on a set of *rules*. The correct format for a rule is as depicted in Fig.3 below.

```
Format style:  rule

target: [prerequisite...]
    command [argument...]
```

Figure 3 – Correct format for a rule in the `mmakefile`. A target is separated from the prerequisites with a colon, the list of prerequisites ends with a newline-character, and the line of program commands with associated arguments must start with a tab-character. A complete rule is always two lines.

Each *rule* consist of two lines, where the first line contains the *target* and its *prerequisites*. The single target comes first on the line, and without any initial whitespace, and is separated from the prerequisites by a colon. Each prerequisite is separated by whitespace, and the list of prerequisites is ended by a newline character. The second line must start with exactly one tab-character, and contains the program command to be executed, with optional arguments.

The prerequisites can either be a specific file necessary for the execution of a rule, or the name of another target with its associated rule which first need to run before the original target can be evaluated. Thus one can formulate dependencies between separate files and programs, to help make sure the program executable always reflect changes in any of the dependency files.

An example of a complete makefile with name `mmakefile` can be seen in Fig.4 below. Naming the makefile to `mmakefile` enables us to run `mmake` without arguments, since this is the default filename for makefiles in our program.

In a makefile with multiple rules, the top rule is denoted the default target and the associated rule will always be evaluated when the program `mmake` is ran without arguments. Thus, here in Fig.4 we have the default first rule with target `mexec` and two dependencies `mexec.o` and `parser.o`. Each of these dependencies are in turn associated with their own rules, and subsequent dependencies. For this particular `mmakefile`, the dependency graph

mmakefile

```
mexec: mexec.o parser.o
    gcc -o mexec mexec.o parser.o

mexec.o: mexec.c parser.h
    gcc -c mexec.c

parser.o: parser.c parser.h
    gcc -c parser.c
```

Figure 4 – Example of a valid `mmakefile` for usage with `mmake`.

can be drawn as in Fig.5 seen below.

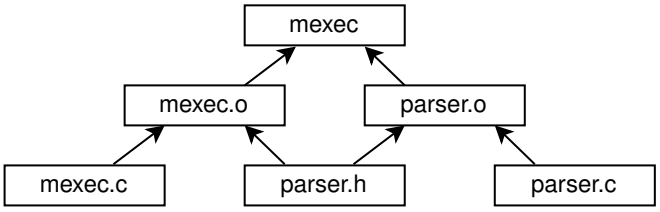

Figure 5 – The dependency graph of the example program `mexec` from Fig.4 above.

Forming this graph in a makefile enables us to propagate changes to one or multiple dependency files upward the dependency tree, such that the final program `mexec` always reflects the latest changes to all dependencies.

As an example, let us run `mmake` with the `mmakefile` from Fig.4. We assume this is the first time running the makefile. Running the program is achieved by executing `mmake` as seen in Fig.6 below.

```
elias@xps13:~/Documents/C-Unix/ou1$ ./mmake
gcc -c mexec.c
gcc -c parser.c
gcc -o mexec mexec.o parser.o
```

Figure 6 – Running the `mmakefile` from Fig.4 above. The executed commands gets printed to the terminal.

Since none of the target files were already existing, all of the rules executed their commands and built the corresponding targets, as seen by the printouts to the terminal. If we now were to execute the same `mmake` command again nothing would happen, since the program senses that no updates have been made in the dependency files, and the targets are already built and up-

to-date. However, if we were to modify one of the dependency files, i.e. by issuing `touch parser.c`, the program would then rebuild this target and propagate the changes up through the dependency graph. We can see exactly this happening in Fig.7 seen below.

```
elias@xps13:~/Documents/C-Unix/ou1$ touch parser.c
elias@xps13:~/Documents/C-Unix/ou1$ ./mmake
gcc -c parser.c
gcc -o mexec mexec.o parser.o
```

Figure 7 – Re-running the `mmakefile` from Fig.4 above. Since we made changes to `parser.c`, it will recompile this target and any target dependent on it, such that changes are propagated upwards in the dependency graph.

As seen in the figure, first the command for the rule associated with `parser.c` gets executed, after which the following rule of linking `parser.o` and `mexec.o` is carried out. As a result, the final executable `mexec` have been rebuilt and now reflects the changes made to `parser.c`, as desired.

3.2 Options and arguments

Once an appropriate `mmakefile` has been written which reflects the dependencies of your program, a small variety of optional flags and program arguments for `mmake` are at your disposal. The synopsis of the program `mmake` is as depicted in Fig.8 seen below.

```
Synopsis:  mmake

Usage: ./mmake [-f MAKEFILE] [-B] [-s] [TARGET...]
```

Figure 8 – Synopsis of the program `mmake`.

The optional flags which the program accepts are:

- `-f MAKEFILE`: Custom filename of makefile
- `-B`: Force build of targets
- `-s`: Silent mode

The optional arguments which the program accepts are:

- `TARGET...`: List of custom targets to evaluate.

The `-f`-flag gives the user the ability to run a makefile with another filename than the default `mmakefile`. However, the other makefile still needs to follow the formatting rules described in the preceding subsection. Directly after the flag `-f` should the custom filename be given, else the program will raise an error.

The `-B`-flag gives the user the ability to force all targets to be built, regardless if `mmake` deemed it to be necessary according to the program logic or not.

The `-s`-flag set the program `mmake` into *slient mode*, i.e. terminal printouts of executed commands gets suppressed.

Additionally, specific targets can be given as arguments to `mmake`, if the user decides to evaluate some other targets than the default target. A single or multiple targets are permitted, as long as they are separated by whitespace and the corresponding rule actually exists in the specified makefile.

If we rename the makefile seen in Fig.4 to `custom_makefile`, we can give an example of running `mmake` with a few flags and arguments, as seen in Fig.9.

```
elias@xps13:~/Documents/C-Unix/ou1$ ./mmake -Bf custom_makefile parser.o
gcc -c parser.c
```

Figure 9 – Demonstration of a few flags and program argument to `mmake`. The flag `-B` forces the target to be built, `-f` specifies a custom filename of the makefile to use, and the last argument chooses a specific target in the makefile to evaluate.

4 Discussion and reflection

This lab was quite a lot of fun to complete, but it also took me way much more time than I had anticipated. At the end I was almost completely out of the allotted time for the lab. I believe what I spent most time on was the recursive component of checking sub-targets, and trying to make sure that the flow of the program was as intended. However, it was very satisfying to finally get it to work. Hopefully for the next lab, I have gotten into the mindset of recursion a lot better such that it will go a bit easier than this one.