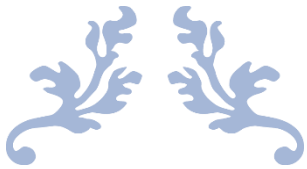

OU2(MMAKE)

Gazi Md Rakibul Hasan
UNIX & C programming
CS: Ens20ghn

OCTOBER 7, 2022

Compilation & Running of the Program

We have created a limited version of the Makefiles for this assignment. In the terminal, it is the duplicate version of the make command. The Makefiles will automate our program's compilation. To automate the compilation, rules for the targets must be defined in the Makefiles. To demonstrate the functionality of Makefiles, we will describe how Makefiles operates when the make command is executed in the terminal. Figure 1 depicts the Makefiles we created to automate the compilation of the program.

```
CC = gcc
FLAGS = -g -std=gnu11 -Werror -Wall -Wextra


all: mmake.o parser.o main.o
     $(CC) $(FLAGS) -o mmake mmake.o parser.o main.o

main.o: main.c
     $(CC) $(FLAGS) -c main.c -o main.o

mmake.o: mmake.c parser.c
     $(CC) $(FLAGS) -c mmake.c -o mmake.o

parser.o: parser.c parser.h
     $(CC) $(FLAGS) -c parser.c -o parser.o

clean:
     rm -rf *.o
```

Figure 1: The Makefiles we have created for our program.

When we will run only make without specifying any target. By default, it will take "all" as the target. We can see that it has some prerequisites, which are **parser.o, mmake.o,** and **main.o**. The Makefiles will build those object (aka .o) files by using the depth-first searching method. It will check the requirements to build those object files and execute the commands. For instance, to build "parser.o" it will go to the "parser.o" target file. Then it will check if there exist rules for the "parser.o". As we can see from Figure 1 that there exist rules for the parser.o, and its prerequisites are "parser.c" and "parser.h" files. Since there exist no rules for the "parser.c" and "parser.h", thus, we can execute the command that exists for the target "parser.o". We will eventually build the "parser.o" file in the directory by executing the command. The same process will be repeated to build other object files such as "main.o" and "mmake.o". Finally, we should have all object files in the directory, which are the prerequisites for the target "all". Afterwards, we will build a link between all object files and create the executable file "mmake". Figure 2 represents how all commands are executed during the creation of the target files in the directory.

Figure 3 illustrates the results of executing the executable file ("mmake") on the computer at Umeå University.

```
itchy:~/Mmake$ make
gcc -g -std=gnu11 -Werror -Wall -Wextra -c mmake.c -o mmake.o
gcc -g -std=gnu11 -Werror -Wall -Wextra -c parser.c -o parser.o
gcc -g -std=gnu11 -Werror -Wall -Wextra -c main.c -o main.o
gcc -g -std=gnu11 -Werror -Wall -Wextra -o mmake mmake.o parser.o main.o
itchy:~/Mmake$
```

Figure 2 shows how commands execute to build the initial target "all".

As shown in Figure 3, we did not specify any additional arguments when running the "mmake" executable. Thus, the rules from the "mmakefile" will be read by default. We have illustrated the "mmakefile" content in figure 4. When we execute the "mmake" executable, it will check the directory for the "mmakefile" file. We will exit the program if we cannot locate the specified file. Afterwards, we will determine whether we can generate a rule from the "mmakfile" file, and if we cannot, we will exit the program. If not, we will continue to the next step.

```
itchy:~/Mmake$ ./mmake
gcc -c mexec.c
gcc -c parser.c
gcc -o mexec mexec.o parser.o
itchy:~/Mmake$
```

Figure 3: The running of our program. If we do not specify any command, then it will read the name "mmakefile" by default. In that case, we did not specify any file in our command.

Since we did not specify any target in the command thus, by default, our initial target will be "all". We have built our program so that it will first check if the target has any rules. If rules exist for the given target, then some prerequisites must exist. We will systematically traverse through all prerequisites of the given target. We will use the depth-first search method to check if those prerequisites have rules. As we can see from figure 4, target "all" has two prerequisites (mexec.o and parser.o). Now we will use the depth-first search method while traversing through the prerequisites of the target "all". When we have reached the prerequisite "mexec.o", we will verify if the" mexec.o" has any rules. We can see that there exist rules for the "mexec.o". It has one prerequisite, which is "mexec.c". Since "mexec.c" has no rules thus, we will not proceed further for the target "mexec.c". Then, we will execute the command which is specified in the "mexec.o". The command's execution will be printed on the standard output. We will repeat the same process to build the "parser.o" file.

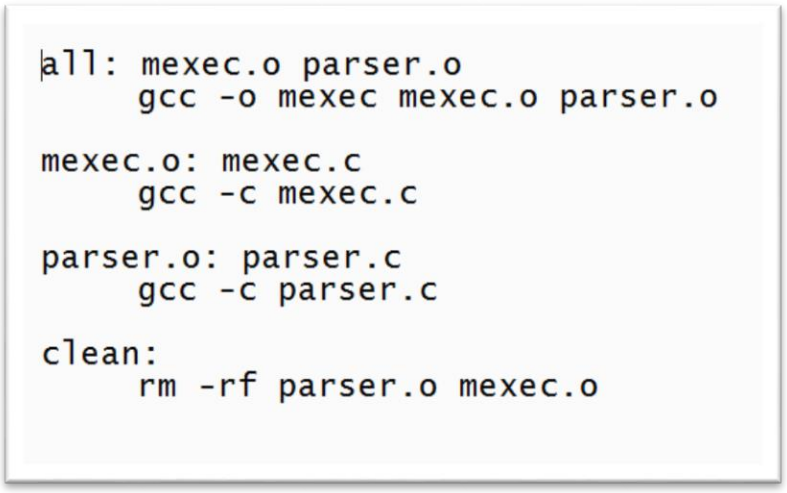

Figure 4: The information contains in the "mmakefile".

After building both 'parser.o' and "mexec.o", there are no more prerequisites to explore. Then we can build the "mexec" executable file by linking all those prerequisites we have created earlier in the directory.

User Usage

In the itchy (Umeå university's computer) or Unix-like operating systems, the user shall run the executable file by writing **./mmake** in the terminal. In that program, we have included several flags such as '-B', '-f' and '-s'. Those flags can be added to the executable file. The user can determine how the program will behave by adding those flags.

When the user inserts '-f' flag along with the executable file, the user must specify a file name that exits the make file rule. The user can perform that task by writing **./mmake -f filename** in the terminal. Suppose the given file does not exist or the given file contents do not satisfy the Makefile's rules. In that case, the program will not proceed further. By default, when the user types**./mmake** in the terminal, the program reads the "mmakefile" It will execute all commands specified by each target.

Suppose the user reruns the executable **./mmake**. In that case, it will not build anything unless the user modifies any of those target files, which includes in the "mmakefile. However, if the user inserts '-B' flag along with the executable file **./mmake -B** in the terminal. Then the program will execute all targets' command that includes in the "mmakefile" even if none of those targets has modified yet. Usually, when the user runs the executed file **./mmake,** it will print each command defined by each target. Nevertheless, if the user does not want these commands will be printed on the standard output, then the user can include "-s". Adding "-s" along with the executable file **./mmake -s** will exclude all command terminations resulting in the standard output.

Discussion

It is fascinating to see how we can create our Makefiles. Thus, I consider this assignment exciting and fun. During the implementation of that program, I encountered bugs. It was challenging to identify those bugs. Sometimes, it took hours to fix these bugs.