CSCi 4061: Intro to Operating Systems
Spring 2016
Instructor: Jon Weissman
Assignment 1: Simple Make
Due: Feb. 17, 11:55 pm

1 Purpose

Make is useful a utility which builds executable programs or libraries from source files based on a *makefile* which includes information on how to build targets (e.g. executable programs or libraries). In this assignment, you will write a simple version of Make (make4061) which 1) reads the makefile, 2) follows the specification in the file and 3) builds the targets in *parallel* (i.e. concurrently) using *fork, exec* and *wait* in a controlled fashion just like standard Make in Linux. Your make4061 will create a dependence graph of all the targets (files) to be compiled at runtime to support control- and data- dependencies. Independent files are compiled in parallel/concurrently while dependent files are compiled once all of its control dependencies have been resolved. You should work in groups of 3.

2 Description

Your make4061 will be responsible for analyzing dependencies of targets, determining which targets are eligible to be compiled (built). That is, each target may depend on other targets which should be evaluated first. As targets in makefile are compiled (built), your Make4061 will determine which targets in the makefile have become eligible to be compiled (built), and continue this process until final target is built. To make this work, you will need to make a DAG (Directed Acyclic Graph) which has *no cycles* from the makefile. For example, figure 1 shows the graph that is created by this makefile. There are two major aspects of this lab. One is the grungy low-level parsing you need to do to build the graph, and the other is the processing of the graph. All C programmers have to eventually deal with the former, even though the main intellectual aspect for us is the second part.

```
1 all: make4061
2
3 make4061: util.a main.o
4         gcc -o make4061 main.o util.a
5
6 main.o: main.c
7         gcc -c main.c
8
9 util.a: parse.o cal.o
10        ar rcs util.a parse.o cal.o
11
12 cal.o: cal.c
13        gcc -c cal.c
14
15 parse.o: parse.c
16        gcc -c parse.c
17
18 clean:
19        rm -rf *.o util.a make4061
```

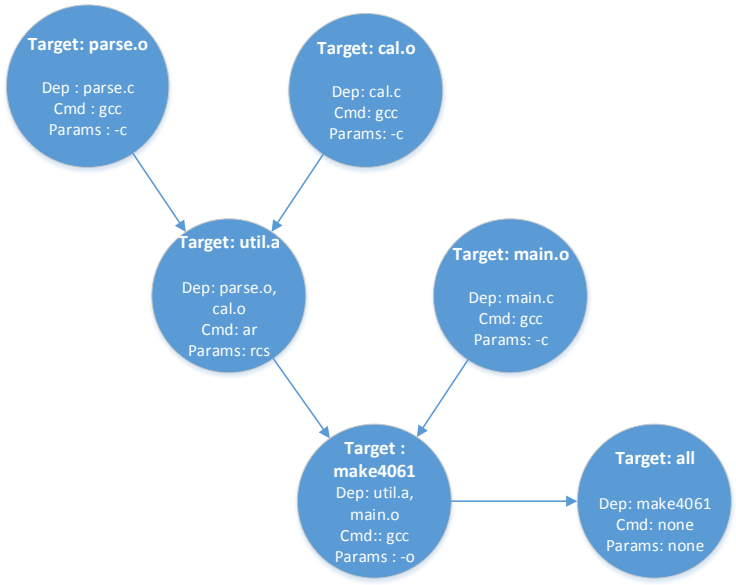

Figure 1: Example makefile and graph

A target becomes eligible for execution once all of its parent targets (that build needed files for this target) have completed their own execution. Your main program will *fork* and *exec* each of the target as they become eligible to

run. In Figure 1, commands for the targets, cal.o, parse.o and main.o, can be executed concurrently because they are not dependent on any other target i.e., they all should start immediately as they are not a child of any other target. After the targets, cal.o and parse.o, are built, then util.a can be built by command 'ar'. Each vertex of the graph represents a target which may contain the following information needed to build the target:

1. The command name and its arguments
2. Dependency information: input target or file(s)
3. And so on depending on your own design

The data-structure for the graph (including each vertex) is up to you. Of course, a DAG is really just a tree. Your make4061 will check whether there are valid input file(s) or target(s) before executing a command. If there are no valid input file(s) or target(s), the program will print an error message and be terminated. After a command finishes, make4061 will check whether the target is successfully built or not. It there is no target file produced, then the program will show an error message and be terminated. For example, in the figure 1, if there is no parse.c, make4061 will not execute the command. If there is an syntax error in parse.c, 'gcc' command will fail to build the target 'parse.o'. In this case, make4061 will print an error message and exit.

To avoid any unnecessary recompilation, your make4061 will check the modification time (timestamps) for the target and input files to see if the target needs to be built. That is, if the target is more recent than input file, then the target is already up-to-date so it doesn't need to be recompiled. For example, in Figure 1 the command for target parse.o will be executed only when the input parse.c is more recent than parse.o. If there is no parse.o file, the command will be executed without checking the modification time. If a user uses the -B option (described below), then make4061 will not check the timestamps and execute all command as specified. Some useful functions are provided in util.c for you to use to help with this.

3 Makefile Format

A makefile will contains multiple targets. The dependence items are either file names or the name of targets specified in another build specification in the same makefile. Here are the rules for the format.

1. The target line starts on the first character of a line.
2. ":" will follow the target and list of dependencies with one or more spaces between them will follow ":".
3. A target is not required to have a dependence (e.g., as in Figure 1, clean:)
4. A target will have at most one command (possibly none).
5. Command line always starts with a tab but not spaces.
6. Commands can be any executable Linux program or shell command.
7. Lines that do not start with a target, or commands that do not start with a tab are not valid, and make4061 will be terminated. That is, if there was any syntax error, your program will be terminated with an appropriate error message.
8. Extra whitespace will be ignored.
9. Lines that start with a "#" will be commented out and ignored.
10. If any error occurs when executing commands, your make4061 will be terminated with an appropriate error message.

11. There will be no cycles among targets. For example, Target1 ->Target2 ->Target3 ->Target1 is not a possible input.

12. Duplicated target name is not allowed.

If anything is unclear, please post your question on the forum to share it with other students.

4 Execution

Your make4061 will support the following options.

1. -f filename: filename will be the name of the makefile, otherwise the default name 'makefile' is assumed.

2. -n: Only displays the commands that would be run, doesn't actually execute them.

3. -B: Do not check timestamps for target and input (i.e. always recompile).

4. -m log.txt: The log from make4061 will be stored on file log.txt (You should use *dup2* for every target. It should produce the approximately the same output as with the command ./make4061 >logfilename (the output may be interleaved differently using *dup2*). *This may take a few lectures to be able to do.*

We will run your make4061 as follows (note that only a single target will be built):

1. ./make4061: This will build the first target found in makefile

2. ./make4061 specifictarget : This will build only the specific target

3. ./make4061 -f yourownmakefile

4. ./make4061 -n

5. ./make4061 -m logfilename

Options can be combined together like below.

./make4061 -f yourownmakefile specificTarget -m output.log

To reduce your burden to parse the makefile, we will provide helpful functions that you may use for the project. Also we will provide some test cases that you can use to test your make4061. You may want to try to test your solution by building *your solution* with make4061!

5 Do's and Dont's

You may **not** use the library call `system` which invokes a shell command externally from your program. You must use `fork` and `exec` instead. You need only execute a single target (i.e. a single graph) with each invovation of your makefile, but you must execute the graph in a concurrent fashion *level-by-level*. That is, the direct children or dependents of a parent node must be `fork`ed concurrently. Once they are all launched, then you should `wait` for them all, before launching any more nodes. A deeper level of concurrency is extra credit. You **MUST** avoid `fork` bombs, and clean up any processes that linger as a result of broken code, especially zombies. The command ps -u $< userid >$ shows your processes and $kill - 9 < processid >$ will get rid of them at the shell. Efficiency is **critical**, that is, your program may iterate repeatedly over your graph data-structure but your program should take seconds to run, not minutes. We **recommend** that you use only static arrays for this assignment (we have included a possible data-structure for your graph node). You are free to use fancy dynamic data-structures, however, if you wish. In general, you are free to use any or none of the provided code.

6 Useful System Calls and Functions

It is highly suggested that you make use of the following system calls or library functions:

`fopen, fgets, fork, dup2, execvp, wait, strcmp, strtok, getopt`
`makeargv` - from R&R page 37.

We highly recommend to check detail of each system call and library function with 'man' command.

7 Simplifying Assumptions

1. Target, command and parameter can only contain alphanumeric characters. Special characters such as \, <, >, & are not allowed. (Thus, there will not be any background processes.)
2. You can assume that each line in the makefile will not be more than 1023 characters.
3. You can assume that there will be no cycles in the graph.
4. You can assume that every command will be on your PATH, thus you do not need to use absolute paths for a command.
5. You can assume that the number of targets will not be more than 10.
6. You can assume that all target will have at most a single command line.
7. You can assume that there are no FLAGS or variables in the makefile like "CFLAGS = -g -Wall". That is, there are only targets and command.
8. You can assume that a user can specify only single target. make4061 parse.o is fine but make4061 parse.o main.o is not.

8 Error Handling

You are expected to test the return value of all system calls to check for error conditions. Also, your main program should check to make sure the proper number of arguments are used when it is executed. If your program encounters an error, a useful error message should be printed to the screen and the program terminated.

9 Documentation

You *must* include a README file which describes your program. It needs to contain the following:

1. The purpose of your program.
2. How to compile the program.
3. How to use the program from the shell (syntax).
4. Any other instructions.

The README file can be short as long as it properly describes the above points. Proper in this case means that a first-time user will be able to answer the above questions without any confusion.

Within your code you should use one or two sentences to describe each function that you write. You do not need to comment every line of your code. However, you might want to comment portions of your code to increase readability.

At the top of your README file and main C source file please include the following comment:

```
/* CSci4061 S2016 Assignment 1
* login:  cselabs_login_name (login used to submit)
* date:  mm/dd/yy
* name:  full_name1, full_name2, full_name3 (for partner(s))
* id:  id_for_first_name, id_for_second_name, id_for_third_name */
```

10 Grading

1. 5% README file
2. 20% Documentation within code, coding, and style
   (indentations, readability of code, use of defined constants rather than numbers)
3. 75% Test cases
   (correctness, error handling, meeting the specifications)
4. Please make sure to pay attention to documentation and coding style. A perfectly working program will not receive full credit if it is undocumented and very difficult to read.
5. Some test cases (sample makefiles) will be provided to you upfront. The test cases for grading will be similar but not necessarily the same. You need to make your own test case as test cases provided may not cover all of the specifications. Thus, please make sure that you read the specifications very carefully. If there is anything that is not clear to you, you should ask for a clarification.
6. We will use the GCC version installed on the CSELabs machines to compile your code. Make sure your code compiles and run on CSELabs.
7. Please make sure that your program works on the CSELabs machines e.g., KH 1-260 (kh1260-xx.cselabs.umn.edu).

11 Extra work (if you are bored – no extra points – you will receive no help on these)

1. Detect and report cycles
2. Exploit full concurrency/parallelism in the graph with a depth-first traversal

12 Deliverables

1. Files containing your code
2. A README file
3. A makefile that will compile your code and produce a program called make4061. Note: this makefile will be used by us to compile your program with the standard make utility.

All files should be submitted on the class moodle website. This is your official submission that we will grade. Please note that future submissions under the same homework title OVERWRITE previous submissions; we can only grade the most recent submission. ONLY one submission is expected for a group. Multiple submissions by different group members particularly if the files differ will make us VERY unhappy. **Communicate effectively, work together, share the load, and submit ONE solution.**