CSCi 4061: Intro to Operating Systems
Fall 2018
Instructor: Jon Weissman
Assignment 1: Simple Make
Due: Oct. 3, 11:55 pm

1 Purpose

Make is a useful utility which builds executable programs or libraries from source files based on an input *makefile* which includes information on how to build targets (e.g. executable programs or libraries). The *makefile* specifies a dependency graph that governs how targets should be built. In this assignment, you will write a simple version of Make (make4061) which 1) reads the makefile, 2) follows the specification in the file to construct the dependence graph and 3) builds the targets using `fork, exec` and `wait` in a controlled fashion just like standard Make in Linux using the graph. Your make4061 will use the dependence graph (**we have provided code to generate it from the input *makefile***) to build any targets (e.g. compile files) in the proper order. The command to finally build the target can be executed once all of its dependencies have been resolved (i.e. those targets have been built and so on). Thus, you will focus only on step 3). You should work in groups of 3. If you are new to makefiles, you will need to read up on them.

2 Description

Your make4061 will be responsible for analyzing dependencies of targets, determining which targets are eligible to be built (i.e. typically to be compiled or executed). That is, each target may depend on other targets. Those other targets must be built first. As targets in makefile are built, your make4061 program will determine which targets in the makefile have become eligible to be compiled (built), and this process continues until final target is built. To make this work, we are providing you with a DAG (Directed Acyclic Graph) based on the contents of makefile. For example, figure 1 shows the DAG that is created by this makefile. There are two major aspects of this lab. One is the grungy low-level parsing that needs to be done to build the DAG (**Note : This has been provided**), and the other is the processing of the DAG. All C programmers have to eventually deal with the former, even though the main intellectual aspect for us is the second part.
A target becomes eligible for execution once all of its parent targets have completed execution. Your main program will use `fork` and `exec` to build each target as they become eligible to run, and `wait` to ensure that a build is finished (or has failed).
The pseudo-code algorithm for building a target T recursively is as follows (where $command$ contains 0 or 1 commands):

build (T):
  for each dependent target T_i that needs to be be built:
    call *build* (T_i)
  execute the *command* for T

In Figure 1, applying the algorithm to building target "all" would yield these these approximate steps:

1. "all" waits for "make4061_test" to finish
2. "make4061_test" waits for "util.a" and then "main.o" to finish one by one.
3. "util.a" waits for "parse.o" and "cal.o" to finish one by one.
4. "parse.o" finishes and then "cal.o" finishes.
5. "util.a" finishes.
6. "main.o" finishes.

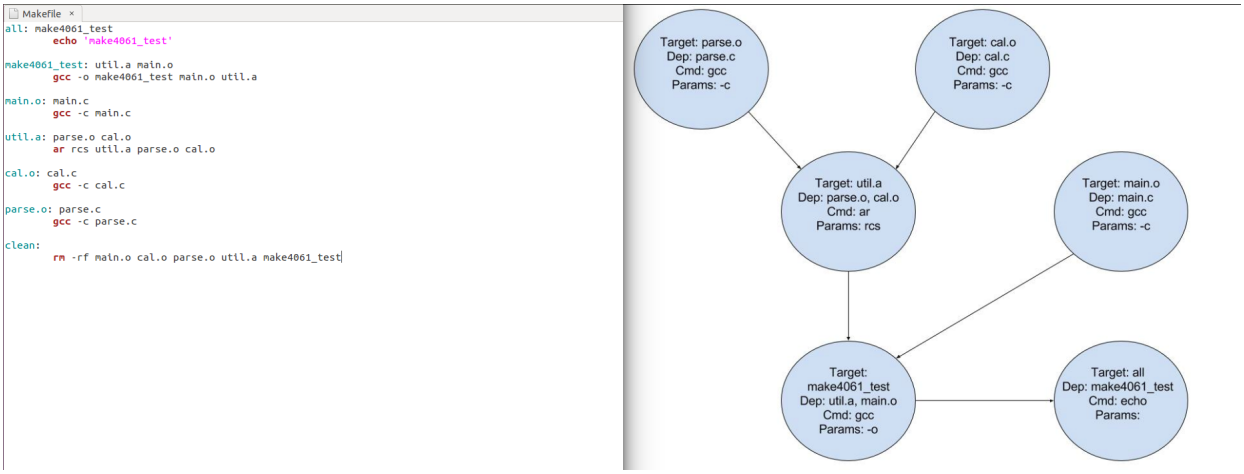

Figure 1: Example makefile and DAG

7. "make4061_test" finishes.

Each vertex of the DAG represents a target which may contain the following information needed to build the target:

1. The target name
2. Dependency information: input target(s) or file(s)
3. The command name and its arguments
4. A status of the vertex, depending on your own design

The data-structure for the DAG (including each vertex) that we are providing is an array representation. Your make4061 program 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. For example, in figure 2, if there is no parse.c, make4061 will terminate. If the target command fails for any reason (e.g. a compiler error in gcc), you can easily detect this using the `wait` child status (**More details in the Error Handling section**), and terminate.

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 2 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. Some useful functions are provided for you to use to help with all of these tasks.

3 Makefile Format

A makefile usually 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.
3. A target is not required to have a dependency (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. Duplicated target name is not allowed.

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

Note: Parsing of the input makefile and creating the associated DAG data structure has been provided. Alternatively, if you want a greater challenge do not look at any provided code and program everything from scratch!

4 Execution

Compile your code with the help of make utility of UNIX. Copy your object file(make4061) to the testcase folder. To test your object file, run: **./make4061 [options]**
To check correctness of your program compare the output of your object file with the one we have provided as solution.

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. specificTarget: specificTarget will be the name of any single target in the Makefile.
3. -h: print usage info.

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

Options can be combined together (in any order) like below.

./make4061 -f yourownmakefile specificTarget

We have provided code for steps 1) and 2) and functions that may be useful for step 3) (see **Purpose**): util.h, util.c, main.c. 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 can **not** use the library call `system` which invokes a shell command externally from your program. You must use `fork`, `exec` and `wait`. You will build targets in the order dictated by the dependency graph (i.e. see the recursive algorithm) using `fork/exec/wait` as appropriate. 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.
Note: If no target is specified on the command-line, build the FIRST target.

6 Getting started

Look at the manpage for make and run the solution code on the test cases. UNDERSTAND how it works. Then, look at the provided starter code, compile and run it. UNDERSTAND IT. Then, look at util.c and check the details of each system call and library function with 'man' command.
`fopen, fgets, fork, execvp, wait, strcmp, strcpy, strtok, getopt`

7 Simplifying Assumptions

1. Target, command and parameter can only contain alphanumeric characters. Special characters such as $\backslash$, $<$, $>$, & 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. As mentioned earlier, if your program encounters an error in processing the makefile, a useful error message (e.g. the name of the current target for which the error occurred) should be printed to the screen and the program terminated.
One crucial/critical place, where you would need to check for errors is in `fork/exec` command(s). Exec may fail due to bad path name, bad memory, out of memory, too many processes and many more. We need to handle problems such as failed command, crashes etc. If a child process fails, the main process (i.e make utility) has failed too and must exit. One solution to check for child process failure is using status in wait. There are multiple ways to achieve this and we are providing one such example using WEXITSTATUS macro:

```
wait(&wstatus);
if (WEXITSTATUS(wstatus) != 0)
{
        printf("child exited with error code=%d\n", WEXITSTATUS(wstatus));
        exit(-1);
}
```

9 Documentation

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

1. Tell us who did what on the program.
2. Any other instructions.

The README file can be short as long as it properly describes the above points. 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 S2018 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: this rubric is tentative and subject to change

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(i.e. 5.4.0) 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 4-250 (csel-kh4250-xx.cselabs.umn.edu). You will be graded on one of these machines.

11 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 canvas site. 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.**

12 Warmup

The simple rule for finishing this project is **Start Early**. In order to get you started we can divide the project in 2 phases. The first phase is understanding the structure of the graph – write code to traverse the graph and print out the per-node information in an easy-to-read manner. In the makefile below shown in Figure 1, every NODE in the DAG has 4 main fields (shown in figure 2) that you should print out:

- TargetName: make4061_test
- DependencyCount: 2
- DependencyNames: util.a, main.o
- Command: gcc -o make4061_test main.o util.a

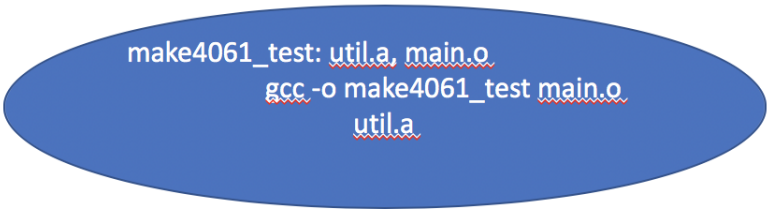

Figure 2: One node in the DAG

We will have your groups check-in with a TA **TO MAKE THIS WORKS IN ABOUT A WEEK.**

13 Testing Strategy Hint

A simple method to tell whether your code works is to compare it's output to that of standard gnu make. This is how we will grade your lab.