CSci 4061: Introduction to Operating Systems

Project 1 - Processes **due: Wednesday October 9th, 2019**

Ground Rules. You may choose to complete this project in a group of up to two students. Each group should turn in **one** copy with the names of all group members on it. The code must be originally written by your group. No code from outside the course texts and slides may be used—your code cannot be copied or derived from the Web, from past offerings, other students, programmer friends, etc. All submissions must compile and run on any CSE Labs machine located in KH 4-250. A zip file should be submitted through Canvas by 11:59pm on Wednesday, Oct 9th. **Note**: Do not publicize this assignment or your answer to the Internet, e.g., public GitHub repo. To share code between team members, create a private repo in github.umn.edu.
Objectives: The focus of this project is to (1) understand the concept of OS processes in C and the functionality of make Unix command. (2) We replicate some functionalities of makefile[1], via fork, exec, and wait system calls.

1 Project Description

A makefile is often used to simplify the compilation and build process of large software tools, via the GNU make [2] command. The software installation targets are primarily UNIX like operating systems. A typical makefile has the following structure [1]

```
target: dependencies
    recipe(s)
```

Given a makefile filename and an optional target as command line arguments, your task is to generate an executable 'mymake', from a C program. It must parse the makefile and build a dependency graph for the (optional) provided target. Also, execute the recipes for the said target, by spawning a new process per recipe, via fork and exec, as determined by the graph traversal. If no target is specified as a command line argument, the first target in the makefile is chosen.

Alternatively, if your first argument is a print flag (-p) followed by the makefile, you must only print the target, dependencies, and recipes in the makefile. A second type of flag (-r) followed by the makefile must print the recipe order of the target. Do not fork/exec the recipes when running either of the flags. More details are described in the following three phases:

1.1 Parsing the input makefile

The makefile consists of multiple targets and each target describes the dependencies that must be satisfied before executing the recipes within a target. Recipes, rules, and commands mean the same thing. In this project makefile, each target can either have multiple dependencies with one recipe in a new line. Targets and dependencies are separated by a colon (:). Multiple dependencies for a target are separated by space. Each recipe for a target must be indented via a single tab (\t, not 2/3/4/8 spaces). Recipes within the same target, must be executed in the order they appear. Blank lines are optional and improve readability. More assumptions about the file are presented later in the section 5

To simplify your job, we will provide a *helper function* that reads the contents of the makefile into a two dimensional char array. The first step is to parse these lines from a char array to accurately determine targets, dependencies, and recipes. We suggest adopting the technique described in *makeargv.c* from the textbook http://usp.cs.utsa.edu/usp/programs/chapter02/. One sample makefile is shown below:

```
all : dep1 dep2
    ls -l

dep1: dep3
    gcc -o file1.o -c file4.c file1.c

dep2:
    gcc -o file3.o -c file3.c

dep3: dep4
    gcc -o file2.o -c file2.c

dep4 :
    mkdir myproject
```

The above makefile with no command line target, runs as ./mymake -p makefile_in. It must print the target name, the dependency count and recipe count for each of the target in the Makefile. Output of one target can look like

```
target 'all' has 2 dependencies and 1 recipe
Dependency 0 is dep1
Dependency 1 is dep2
Recipe 0 is ls -l
```

Similarly, you must print the remaining 4 targets, their respective dependencies and recipes to get full credit for this phase.

1.2 Determining the order of recipes

The makefile 1.1 when run with the recipe flag option, must print the order of recipes, such as running ./mymake -r makefile_in on command line. It must pick *all* and look for recipes in the order - dep4, dep3, dep1, dep2. Finally, it prints the following lines:

```
mkdir myproject
gcc -o file2.o -c file2.c
gcc -o file1.o -c file4.c file1.c
gcc -o file3.o -c file3.c
ls -l
```

Running mymake -r makefilename will print all the recipes in order of execution, and no targets must be passed on the commandline.

1.3 Fork and exec the recipes.

Using the data structures that model the graph, execute the recipes by forking a new process per recipe. Your program should determine which recipes are eligible to run, execute those recipes, wait for any recipe to finish, then repeat this process until all recipes finished executing.
Instead, if a project is run with a specific target, say dep2, ./mymake makefile_in dep2, your program must only print and then fork/exec one recipe: gcc -o file3.o -c file3.c

2 Implementation:

We will provide an example framework to get started and examples to test the code. However, you are free to use your own implementation, without following the given structure. In the folder *pa*1, the *src* folder contains the code, *test* contains the test cases . The data structure used to store information about the target in desired 'container' data structure looks like

```
typedef struct target_block {
  char *name; //Target name
  char *depend[MAX_DEP]; // dependencies of target
  char *recipe[MAX_RECIPES_PT]; // recipes per target
  unsigned char dep_count; // number of dependencies in target
  unsigned char recipe_count; // number of recipes in target
  unsigned char visited; // used in DFS
} target;
```

The dependency graph uses the result of topological sort [3] - an application of Depth First Search algorithm. Given a set of targets, dependencies, and recipes, one must construct a directed graph by determining the nodes and edges as targets and dependencies, respectively. Alternatively, a less modular deisgn involves inserting the targets into a stack based on the DFS traversal, avoiding building a graph.

3 Execution:

There are three ways to execute the project via
$ ./mymake filename [target]
$ ./mymake [-p] filename
$ ./mymake [-r] filename

Your executable can take upto two command line arguments. The filename argument is mandatory. If the optional arguments are provided, they must be in fixed positions. Any other variations must error out.

4 Testing

As described in 2, to test your code against the provided test cases within the framework, run $ make -i test-main. The -i flag ignores the error return value and lets you test the whole suite. We

will use more tests cases than the ones provided. If you have not attempted extra credit, discussed later, you must modify the makefiles to suit your code.

5 Assumptions:

To simplify the project, you can assume the following about the makefile:

- The makefile is well formed and has no circular dependencies.
- It doesnt use make variables (denoted via $), macros, or wildcards.
- It can have a maximum of 128 lines and each line is upto 128 characters long
- Each target can have upto 8 dependencies and one recipe. This assumption is relaxed, if you are attempting extra credit.
- There can be no more than 128 targets in the makefile. Targets having 0 dependencies and 0 recipes, simulatenously, do not exist. However, targets with 0 recipes are possible.
- Recipes do not spread across lines
- We limit the executables to those found in default PATH environment of the CSELAB machines. Sample executables are Linux commands such as ls, cat, pwd, mkdir, or building an executable via gcc.
- Each makefile has a single dependency tree.
- You are free to make any reasonable assumptions not stated in this document, but do mention them early in your README file.

6 Extra credit:

Unlike the makefile described earlier, one can specify multiple recipes for a target. Multiple recipes in different lines, within the same target, are executed in the order they are read. Recipes order within the same line are from left to right. Also, multiple recipes can be solved in parallel by separating them with a comma (,). For example, assume the following recipes in your makefile named makefile_ec

```
all : dep1 dep2
    ls -l

dep1:  dep3
    gcc -o file1.o -c file4.c file1.c

dep2:
    gcc -o file3.o -c file3.c

dep3: dep4
```

```
    gcc -o file2.o -c file2.c
    gcc -o file4.o -c file4.c

dep4 :
    mkdir myproject, pwd
    cat file1.c
```

By executing this makefile with no targets as $ ./mymake makefile_ec. The program must pick *all* and look for dependencies in the order — dep4, dep3, dep1, dep2. The recipes are executed as:

```
mkdir myproject
pwd
cat file1.c
gcc -o file2.o -c file2.c
gcc -o file4.o -c file4.c
gcc -o file1.o -c file4.c file1.c
gcc -o file3.o -c file3.c
ls -l
```

For dep4, do not execute cat file1.c until both the previous commands finished executing. The parent must use *wait* system call on both the commands to avoid zombie process. If your group chooses to attempt the extra credit for an additional 10% of the project grade, you must correctly handle both parallel execution of recipes and multiple recipes per target, within the same program.

7 Deliverables:

One student from each group should upload to Canvas, a zip file containing their C source code files, a makefile, and a README that includes the following details:

- The purpose of your program
- How to compile the program
- What exactly your program does
- Any assumptions outside this document
- Team names and x500
- Your and your partners individual contributions
- If you have attempted extra credit

The README file does not have to be long, but must properly describe the above points. Proper in this case refers to – first-time user can answer the above questions without any confusion. Within your code you should use one or two comments 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 answer why, rather than how you implement the said code. At the top of your README file and main C source file please include the following comment:

```
/*test machine: CSELAB_machine_name
* date: mm/dd/yy
* name: full_name1, [full_name2]
* x500: id_for_first_name, [id_for_second_name]
*/
```

8 Grading Rubric:

1. 10% Correct README contents
2. 10% Code quality such as using descriptive variable names, modularity, comments, indentation. You must stick to one style throughout the project. If you do not already have a style for C programming, we suggest adopting K&R style [4].
3. 15% for using meaningful data structures and appropriate error checking and handling
   (a) 10% No program crashes or segfault
   (b) 5% uses appropriate data structures
4. 20% Parse the makefile accurately - determined via -p flag
5. 15% Implement the dependency graph algorithm for identifying recipe order – determined via -r flag
6. 30% Correctly call fork, exec, wait system calls not leading to fork bombs or zombie processes.
7. 10% Extra credit

If your code passes the test cases, you will get 75% of the credit. The remaining 25% depends on the implementation quality and documentation, as described in points 1, 2 & 3b.

9 References:

1. Makefile rules: `https://en.wikipedia.org/wiki/Makefile#Rules`
2. GNU Make: `https://www.gnu.org/software/make/manual/make.html`
3. Topological sorting: `https://en.wikipedia.org/wiki/Topological_sorting`
4. K&R Style guide: `https://en.wikipedia.org/wiki/Indentation_style#K&R_style`
5. AND operator in make `https://www.gnu.org/software/make/manual/make.html#Execution`