UNIVERSITY OF WISCONSIN
Computer Sciences Department

CS 537
Fall 2020

Barton Miller

Programming Assignment #3: Make

Handed out: Friday, October 16
~~**Due: Wednesday, November 4, 5pm**~~
Now due: Thursday, November 5, 12:30pm

There are several goals for this assignment.

1. Learn about creating new processes, loading new programs, and waiting for processes to complete in Linux.
2. Get experience with poorly formed input and handling the errors gracefully. (Defensive programming.)
3. Learning a few more useful Linux kernel calls, such as `fstat()`.

Your assignment is to write a simple version of the `make` command. Your version of `make`, called `537make` will be executed from a command line. It will read a a "makefile" and following the specifications in the makefile as to which commands need to be executed. So, you will implement a small but quite useful subset of the standard Linux `make` program.

The Makefile

Your `537make` will look first for a file called "makefile"; if that is found, then it will use that file to run. If it does not find "makefile", then it will look for a file called "Makefile". If neither file is found, then report the error and exit.

A makefile will contain a sequence of build specifications of the form:

```
target: dependence1 dependence2 ... dependencen
        command1 ...
        command2 ...
        ...
        commandm
```

Here is an example makefile, not unlike the one you probably used for Program 1:

```
537ps:  537ps.o readproc.o parseopts.o output.o
        gcc -o 537ps 537ps.o readproc.o parseopts.o output.o
        echo built successfully....

537ps.o: 537ps.c 537ps.h readproc.h parseopts.h
        gcc -c -Wall -Wextra 537ps.c

readproc.o: readproc.c readproc.h
        gcc -c -Wall -Wextra readproc.c

parseopts.o: parseopts.c parseopts.h
        gcc -c -Wall -Wextra parseopts.c

output.o: output.c
        gcc -c -Wall -Wextra output.c
```

The dependence items are either file names or the name of targets specified in another build specification in the same makefile. The commands are arbitrary Linux shell commands.

Makefile Format

A few simple requirements for the makefile format:

1. A target line always begins in the first column (i.e., starts on the first character of a line).
2. The target (which is is usually the name of a file being created by one of the commands in the build specification) is followed by a ":" character and then a list of dependence names, each separated by white space (any number of spaces or tabs).
3. A dependence is either the name of another target or the name of a file. It is an error if there is no file or target that matches the dependence.
4. If there is a cycle in the chain in any of the dependences, then that is an error.
5. A command line always starts with a tab character (not spaces).
6. The list of commands for a target ends either when a new target starts or at the end of the files.
7. Blank lines are ignored and can appear in the middle of a list of commands. Blank means an empty line or one with only white space.
8. A line that begins with a "#" as the **first** character on the line is a *comment line*. The rest of the line is ignored. Comment lines are ignored and can appear in the middle of a list of commands. Note that if a comment line is too long, it is reported as an error.
9. Lines that contain a null (zero) byte within the line are invalid and reported as an error.
10. Non-blank lines must begin with a target name or tab; otherwise they are invalid and reported as an error.
11. Any command that exits with an error (a non-zero completion code) terminates the make process.
12. When you find an error while parsing the makefile, you will print an error message to stderr. This error message will look like:

    ```
    10: <error message>: "blah blah blah"
    ```

    where "<error message>" is an informative error message and "10" is the line number in the makefile of the bad line and "blah blah blah" is the contents of the bad line. After printing the error message, your program will exit.
13. When you find an error while executing lines in your makefile, such as file not found, you will print an error message to stderr.
14. On any error, after printing the error message, your program should exit.
15. Set your maximum line length to 4K (4096), including terminating null byte.

Each line that has a command will specify a program name and and zero or more arguments; it is of the form:

```
cmd arg1 ... argk
```

Each of these items is separated by one more more blank characters (spaces or tabs). There is one command per input line. Your program will fork a child process and then overlay itself (exec) the command in the file named by "cmd". The 537make (parent) process will wait for the child process.

The command line is parsed by your program into a list of character strings, one for each argument (including the command name). These arguments are passed as parameters to the exec command (you'll probably want to use `execvp()`).

Make sure that your program can handle input lines that have very long command names and arguments. You must also be able to deal gracefully with command lines that are arbitrarily long; you must be able to gracefully reject them.

Make Rules

1. If the target is a file name, its time is the modification time of the named file. If the target is not a file name, its time is zero.
2. The commands in the specification are run when a dependence is **out of date**. A dependence is out of date when:
    1. The dependence name is a file and that file's modification time is more recent then the time of the target.
    2. The dependence name is another target and that target was out of date.
    3. If there are no dependences for a target, then it is considered out of date and the commands in the specification **always** run.

Running 537make

You can run your program simply by typing:

```
% 537make
```

In this case, you will make the first build specification found in the file. Alternatively, you can (using the example makefile in the previous section), type:

```
% 537make readproc.o
```

In this case, you will make the build specification labeled "readproc.o".

UNIX/Linux Manual Pages

For this assignment, there are several new Linux kernel calls that you will use.

You will create a new process with `fork()`, as shown in Lecture 3 of the class notes. You can get the details of this command from the Linux manual:

```
% man 2 fork
```

You will load a new program into the child process by using a varient of the `exec` command, probably using `execvp()`. You can get the details of this command from the Linux manual:

```
% man 3 execvp
```

There are several varients of "exec" and you will need to use one of the varients that has a "p" in its name to make sure that you use the PATH environment variable to search the right directories for the file that contains the command.

You will need to check the modification time on a file to see if a target is out of date. For this operation, you can call `stat()` on a file path name or `fstat()` on an open file. You can get the details of these commands from the same Linux manual page:

```
% man 2 stat
```

Program Structure

As with the previous assignment, it is important to have a clean, modular design; you will need to develop clean interfaces for each component. And, of course, each component will be in a separately compiled file, linked together as the final step. `

Some suggestions for modules in your design:

- *Build specification graph:* This module has functions for both building the graph that represents the dependences between build specifications, and traversing the graph in a bottom-up order to evaluate the specifications (effectively a post-order traversal).
- *Text parsing:* This module contains functions that help you parse lines in the makefile. It splits a line into an array of strings, checking whether the line begins with a tab or regular character, and filters out blank lines.
- *Build specification representation:* This module contains the basic build specification abstraction. It allows you to create, update, and access a build specification.
- *Process creation and program execution:* This module is responsible for running each build command in a new process, waiting for its completion, and getting the return code.

Testing Your Program

As with the first assignment, you will want to test your program in pieces before assembling it. This will simplify testing of each part and shorten the time to get the whole program running.

In addition, you will need to run your program on an input file (makefile) that contains complete random input. The goal of doing this is to help to make sure that your error handling is thorough and robust.

Your program should not crash, no matter how weird the input. For example, suppose that the makefile has zero-value characters (bytes) or lines that have 1,000,000 characters? As a result, you may not be able to use the input routines that first occur to you.

In addition, you will need to demonstrate your `537make` on a makefile that builds your `537make` program.

Deliverables

You can work individually or in a group of two. In either case, you will turn in a single copy of your program, clearly labeled with the name and logins of both authors.

You will turn in your programs, including all .c and .h files and your makefile. Also include a README file which describes a little bit about what you did for this project.

Note that you must run your programs on the Linux systems provided by the CS Department. You can access these machines from the labs on the first floor of the Computer Sciences Building or from anywhere on the Internet using the ssh remote login facility. If you do not have ssh on your Windows machine, you can download this client:

> SSHSecureShellClient-3.2.9.exe

Handing in Your Assignment

Your CS537 handin directory is `~cs537-1/handin/`*`your_login`* where *`your_login`* is your CS login. Inside of that directory, you need to create a `proj3` subdirectory (unless the TAs created one for you already).

Copying your files to this directory is accomplished with the `cp` program, as follows:

```
shell% cp *.[ch] makefile README ~cs537-1/handin/your_login/proj3
```

You can hand files in multiple times, and later submissions will overwrite your previous ones. To check that your files have been handed in properly, you should list `~cs537-1/handin/your_login/proj3` and make sure that your files are there. The handin directories will be closed after the project is due.

Whether you are working individually or in pairs, you should:

1. Submit only one copy of your code.
2. Create a file called partner.txt in each of of your `proj3` directories. It should have a line in the file for each person who worked on the code (so, 1 or 2 lines). Each line will have you name, CS login and NetID.

Original Work

This assignment must be the original work of you and your project partner. Unless you have explicit permission from Bart, you may not include code from any other source or have anyone else write code for you.

Use of unattributed code is considered plagiarism and will result in academic misconduct proceedings (and "F" in the course and a notation on your transcript).

Extra Credit

You can add a couple of features to your `537make` program for extra credit:

`-f`
: As described above, you can allow the user to specify the name of the makefile. (1.0 point)

< and **>**
: You can implement I/O redirection for standand input and standard output for each command line. (2.0 points)

Of course, make sure that you fully implement and test the required features in this assignment before you consider doing extra work.

Last modified: Tue Nov 3 14:36:55 CST 2020 by bart