CS360 Lab #4 -- Fakemake

- James S. Plank
- CS360

What you submit

You submit the file **fakemake.c**. The TA's will copy this to their own directory on the lab machines, and then compile with:

```
UNIX> make -f /home/jplank/cs360/labs/Lab-4-Fakemake/lab-makefile
```

You may use Libfdr for this assignment.

Description of fakemake

Now we get to a more fun assignment. The goal is to write a restricted and pared-down version of **make(1)**. This will give you some practice using **stat(2v)** and **system(3)**.

Your job is to write the program **fakemake**. Like **make**, **fakemake**'s job is to help you automate compiling. Unlike **make**, **fakemake** limits itself to making one executable, and assumes that you are using **gcc** to do your compilation.

The syntax of **fakemake** is as follows:

```
fakemake [ description-file ]
```

If no **description-file** is specified, then it assumes that the description file is the file **fmakefile**. Obviously, if the description file doesn't exist, then the program should exit with an error.

Each line of the description file may be one of six things:

- A blank line (i.e. a line containing only white space).
- A specification of C source files that are used to make the executable:

  ```
  C list-of-files
  ```

  All files thus specified should end with **.c**. Multiple files can be separated by whitespace. The list may be empty, and there can be multiple lines starting with **C**.
- A specification of C header files that may be included by any source file:

  ```
  H list-of-files
  ```

  The **H** list is formatted just like the **C** list, and there can be multiple lines starting with **H**.
- A specification of the name of the executable:

  ```
  E name
  ```

  There can be only one executable name; thus, there can be only one line starting with **E**. It is an error if there is no **E** line in the entire description file.
- A specification of compilation flags:

```
F flags
```

These flags will be included right after **gcc** whenever it is called. As before, the flags are separated by whitespace, and there can be multiple **F** lines. This can be empty as well.

- A specification of libraries or extra object files for linking

```
L list-of-libraries
```

As before, multiple files should be separated by whitespace. The list may be empty, and there can be multiple lines starting with **L**. This list is included at the end of the final **gcc** command that makes the executable.

What **fakemake** does is compile all the **.c** files into **.o** files (using **gcc -c**), and then compile all the **.o** files into the final executable. Like **make**, it doesn't recompile a file if it is not necessary. It uses the following algorithm to decide whether or not to compile the **.c** files:

- If there is no **.o** file corresponding to the **.c** file, then the **.c** file must be compiled (with **-c**).
- If there is a **.o** file corresponding to the **.c** file, but the **.c** file is more recent, then the **.c** file must be compiled (again with **-c**).
- If there is a **.o** file corresponding to the **.c** file, and *any* of the **.h** files are more recent than the **.o** file, then the **.c** file must be compiled.

If the executable file exists, and is more recent than the **.o** files, and no **.c** file has been recompiled, then **fakemake** does not remake the executable. Otherwise, it does remake the executable (using **gcc -o**).

Obviously, if a **.c** or **.h** file is specified, and it does not exist, **fakemake** should exit with an error. If there are any compilation errors mid-stream, **fakemake** should exit immediately. The order of files should be the order specified (this is for compilation and for constructing the final executable).

Example

For example, get into a clean directory and then type

```
UNIX> cp ~jplank/cs360/labs/Lab-4-Fakemake/* .
UNIX> ls
f.c             f.h             f2.c            makefile        mysort.fm
f.fm            f1.c            lab.html        mysort.c
UNIX> make
gcc -c -g f.c
gcc -c -g f1.c
gcc -c -g f2.c
gcc -g -o f f.o f1.o f2.o
gcc -c -g -I/home/jplank/cs360/include mysort.c
gcc -g -o mysort mysort.o /home/jplank/cs360/objs/libfdr.a
UNIX> f
This is the procedure F1 -- in f1.c
This is the procedure F2 -- in f2.c
UNIX> mysort < f.c
  f1();
  f2();
main()
{
}
UNIX> make clean
rm -f core *.o f mysort
UNIX> ls
f.c             f.h             f2.c            makefile        mysort.fm
f.fm            f1.c            lab.html        mysort.c
UNIX>
```

So, this directory contains source code for two programs. The first, **f**, is made up of three C files: **f.c**, **f1.c** and **f2.c**, and one header file: **f.h**. The second is **mysort.c** from the **Rbtree-1** lecture. The **makefile** contains a specification of how to make these programs using **make**. The file **f.fm** is the **fakemake** file for making **f**, and **mysort.fm** is the **fakemake** file for making **mysort**. Try it out, using the **fakemake** executable in **/home/jplank/cs360/labs/Lab-4-Fakemake/fakemake**:

```
UNIX> ~jplank/cs360/labs/Lab-4-Fakemake/fakemake
fakemake: fmakefile No such file or directory
UNIX> ~jplank/cs360/labs/Lab-4-Fakemake/fakemake f.fm
gcc -c -g f.c
gcc -c -g f1.c
gcc -c -g f2.c
gcc -o f -g f.o f1.o f2.o
UNIX> touch f.c
UNIX> ~jplank/cs360/labs/Lab-4-Fakemake/fakemake f.fm
gcc -c -g f.c
gcc -o f -g f.o f1.o f2.o
UNIX> rm f
UNIX> ~jplank/cs360/labs/Lab-4-Fakemake/fakemake f.fm
gcc -o f -g f.o f1.o f2.o
UNIX> touch f.h
UNIX> ~jplank/cs360/labs/Lab-4-Fakemake/fakemake f.fm
gcc -c -g f.c
gcc -c -g f1.c
gcc -c -g f2.c
gcc -o f -g f.o f1.o f2.o
UNIX> touch f.h
UNIX> touch f.o f1.o
UNIX> ~jplank/cs360/labs/Lab-4-Fakemake/fakemake f.fm
gcc -c -g f2.c
gcc -o f -g f.o f1.o f2.o
UNIX> ~jplank/cs360/labs/Lab-4-Fakemake/fakemake f.fm
f up to date
UNIX> f
This is the procedure F1 -- in f1.c
This is the procedure F2 -- in f2.c
UNIX> ~jplank/cs360/labs/Lab-4-Fakemake/fakemake mysort.fm
gcc -c -g -I/home/jplank/cs360/include mysort.c
gcc -o mysort -g -I/home/jplank/cs360/include mysort.o /home/jplank/cs360/objs/libfdr.a
UNIX> mysort < f.c
  f1();
  f2();
main()
{
}
UNIX> rm f.h
UNIX> ~jplank/cs360/labs/Lab-4-Fakemake/fakemake f.fm
fmakefile: f.h: No such file or directory
UNIX>
```

As you can see, **fm** works according to the above specification. It only recompiles modules when it needs to. When you're in doubt about what your **fakemake** should do, see what **/home/jplank/cs360/labs/Lab-4-Fakemake/fakemake** does and emulate its behavior.

The Grading Script

The grading script works by running two scripts -- **your-script.sh** and **correct-script.sh** Each script copies files to the current directory from the lab directory, compiles them, maybe deletes some files or modifies their times, then calls fakemake.

If your program isn't working like mine, modify **your-script.sh** to perform a "ls -l --full-time" listing right before the **fakemake** call. Here's an example. First, I run the gradescript on example number 3. It runs correctly, but I still want to see what **your-script.sh** does:

```
UNIX> ~jplank/cs360/labs/Lab-4-Fakemake/gradescript 3
Problem 003 is correct.

Test: sh -c 'sh your-script.sh > tmp-003-test-stdout.txt 2> tmp-003-test-stderr.txt'

Correct output generated with : sh -c 'sh correct-script.sh > tmp-003-correct-stdout.txt 2> tmp-003-c
UNIX> cat your-script.sh
ge=/home/jplank/cs360/labs/Lab-4-Fakemake/Gradescript-Examples
cp $ge/onefile.c .
gcc -c onefile.c
sleep 1
touch onefile.c
rm -f onefile
cp $ge/001.fm fmakefile
if ./fakemake; then
  ./onefile
fi
UNIX> sh your-script.sh
gcc -c onefile.c
gcc -o onefile onefile.o
With a taste of your lips
I'm on a ride
You're toxic
I'm slipping under
With a taste of poison paradise
UNIX>
```

If you can read the script -- it copies **/home/jplank/cs360/labs/Lab-4-Fakemake/Gradescript-Examples/onefile.c** to the current directory, compiles it to **onefile.o**, then sleeps for a second and makes sure that there is no file **onefile** in the current directory. It then copies the fake-makefile **001.fm** from **/home/jplank/cs360/labs/Lab-4-Fakemake/Gradescript-Examples** and runs **fakemake**. Since **onefile.c** should be one second newer than **onefile.o**, **fakemake** should recompile **onefile.c** to **onefile.o**, and then compile **onefile.o** to **onefile**. It should exit correctly, and then the script runs **onefile**, which, like all the example programs, prints some random lines from *Toxic* by Britney Spears (a surprisingly good song, if you ask me) in a random order.

Below, I modify **your-script** so that it does "ls -l --full_time" on the **onefile** programs:

```
UNIX> vi your-script.sh
UNIX> cat your-script.sh
ge=/home/jplank/cs360/labs/Lab-4-Fakemake/Gradescript-Examples
cp $ge/onefile.c .
gcc -c onefile.c
sleep 1
touch onefile.c
rm -f onefile
cp $ge/001.fm fmakefile
ls -l --full-time onefile*                # this is the new line
if ./fakemake; then
  ./onefile
fi
UNIX> sh your-script.sh
-rw-r--r-- 1 jplank loci  246 2011-02-09 11:05:36.000000000 -0500 onefile.c
-rw-r--r-- 1 jplank loci 1880 2011-02-09 11:05:35.000000000 -0500 onefile.o
gcc -c onefile.c
gcc -o onefile onefile.o
With a taste of your lips
I'm on a ride
You're toxic
I'm slipping under
With a taste of poison paradise
UNIX>
```

You can see from the long listing that **onefile.o** is older than **oldfile.c** by one second.

The test files from 6 through 25 test two-file compilations, and the remainder test header files, libraries, etc.

Details

Obviously, you'll have to use **stat()** to test the ages of programs. The **st_mtime** field of the **struct stat** should be used as the age of the program.

To execute a string, you use the **system()** procedure. It executes the given string as if that string is a shell command (**sh**, not **csh**, although it shouldn't matter). If it returns -1, then it couldn't execute the command. Otherwise, it returns that **exit()** value of the command. You should assume that if a program exits with a non-zero value, then there was an error, and you should stop compiling (i.e. your **fakemake** should exit).

Strategy

It's my hope that you don't need these sections on strategy too much, but to help you out, here's how I wrote this program.

- Wrote the code to figure out the name of the description file. Made my makefile, and tested the code.
- Wrote the main loop to read the description file (using the fields library). This loop just calls **get_line** and prints each line to stdout.
- Wrote the code to recognize blank lines.
- Wrote code to recognize the **C** lines. All other lines are ignored.
- Wrote a subroutine to read the **C** files into a dllist. After reading the entire description file, I print out the contents of this dllist.
- Wrote code to recognize the **H** lines, and used the same subroutine as above to read the **H** files into another dllist. Tested this code.
- Wrote code to recognize and read the **L** lines -- this used the same subroutine and the filenames went into a third dllist.
- Wrote code to read in the executable name. Tested this code.
- Wrote code to recognize and read in the **F** lines. This again used the same subroutine as the **C**, **H**, and **L** lines. The flags are read into another dllist.
- Wrote code to flag an error for any unprocessed line. Also flagged an error if there is no **E** line.
- Wrote code to process the header files. This code traverses the **H** dllist and calls **stat** on each file. It flags an error if the file doesn't exist. Otherwise, it returns the maximum value of **st_mtime** to the **main()** program.
- Wrote code to start processing the C files. This code traverses the **C** dllist and calls **stat** on each file. It flags an error if the file doesn't exist. Otherwise, it looks for the **.o** file. If that file doesn't exist, or is less recent than the C file or the maximum **st_mtime** of the header files, then I printed out that I need to remake that file. I tested this code extensively.
- Wrote code to actually remake the **.o** files. This means creating the ``**gcc -c**'' string and printing it out. Once this looked right, I had the program call **system()** on the string.
- Finished up the C file subroutine by having it return whether any files were remade, or if not, the maximum **st_mtime** of any **.o** files, to the **main()** program.
- Wrote code to test whether or not the executable needed to be made.
- Finally, I wrote code to make the executable. First I wrote code to create the ``**gcc -o**'' string, and then I printed out the string. After testing that, I had the program call **system()** on the string.
- Put finishing touches on the program, and did more testing.