HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2020
ASSIGNMENT 1

This assignment is to be done individually. It is recommended to do Lab 1 before this assignment.

It is important that you submit the assignment in time. **See the deadline in the Student Portal.**

The assignment consists of four parts, described in the sections below. Start by creating a directory for this assignment, and put the resulting files for each part in subdirectories `part1`, `part2`, `part3`, and `part4`. When you are ready to submit your assignment, package your files in the way described in Section 5 below.

1. Part 1

In this part, you create a small makefile that only outputs some text using the `echo` command.

First make sure you understand how the `echo` command works. Try using it, type e.g. `echo Hej hej` and see what happens. It is a very primitive command, all it does is to give as output the same thing that you gave it as input.

As we saw in Lab 1, a makefile can contain a number of rules, where each rule is defined in this way:

```
target: dependency1 dependency2 dependency3
<TAB>command-to-create-target
```

Note: in Lab 1 there was an example makefile that can be useful as a starting point when creating makefiles, but in this particular assignment you should *not* do that. Here, you are to create a small makefile step by step starting from an empty file. You should write the makefile here completely by yourself, to help making sure that you understand the meaning of each part of the makefile. Each step is explained below.

Although the instructions given here are probably enough, you can if you want also look something up in the GNU make manual:
http://www.gnu.org/software/make/manual/html_node/

Start by creating a makefile with just one target without any dependencies. Let the target be called `hello` and let the command-to-create-target be `echo Hello`. So this first makefile should now have just two lines in it. Let the filename be

Date: January 19, 2020.

`Makefile` with capital M (make accepts both names, `makefile` or `Makefile`, but in this assignment we want it to be called `Makefile`).

Check that the makefile works by running `make`. It should give the following output:

```
echo Hello
Hello
```

The first line is printed because `make` by default prints each command it is executing. So we see first the command, and then the output that resulted from executing the command. Note that some commands have no output at all (e.g. when you compile some file with gcc and there are no errors or warnings). In such cases we only see the command.

Now expand your makefile so that it has 3 targets called `hello1`, `hello2`, and `hello3`. Let the command for each target be an echo command including the number, e.g. `echo Hello2` for the `hello2` target. Now you should have a makefile with 6 lines.

Test the makefile by running `make`, and verify that you can choose which target you want by giving that as an input argument to `make`, e.g. `make hello2`. Note that when not specifying which target you want, `make` will by default choose the first target.

Makefiles can also use variables. See for example the variables used in the `makefile-1` in Lab 1. There variables called `CC`, `LD`, `CFLAGS` etc are used.

Add a line in the beginning of your makefile defining a variable called `NAME` and set its value to your own name. So if your name is Kim, the line should look like this:

```
NAME = Kim
```

At this point your makefile should consist of 7 lines; one line with the NAME variable followed by 6 lines defining the three targets.

To use the value of a variable in a makefile, you write `$(VARIABLENAME)` so for the NAME variable it will be `$(NAME)`.

Add `$(NAME)` in the line for the `echo` command for each target in your makefile, e.g. change `echo Hello2` to `echo Hello2 $(NAME)`.

Test your makefile again. Now the value of the NAME variable should be used, so the output should look like this (but with your name instead of Kim):

```
echo Hello1 Kim
Hello1 Kim
```

When variables are used in a makefile, we can set the values of those variables from outside, when executing the `make` command. This is done by giving VARIABLE-NAME=value as input to `make`, e.g. like this:

```
make NAME=Maria hello3
```

When a variable is given a value in that way, that value will be used instead of the value written in the makefile. Check that your makefile allows you to set the

NAME variable in that way. Once you have checked that, part 1 of this assignment is done. Leave your final makefile as the result of part 1, in the part1 directory.

2. Part 2

In this part, you are to create a makefile for a small C program consisting of the source files in the part2 directory. Use the filename `Makefile` here also.

If we want to compile and link the program manually, without using a makefile, the necessary commands are the following:

```
gcc -c stuff.c
gcc -c themainprog.c
gcc -o pyramid stuff.o themainprog.o
```

The first two commands are for compiling stuff.c and themainprog.c to create stuff.o and themainprog.o, respectively, and the third command is to link those object files creating the final executable file called "pyramid".

Write a makefile that has the three targets stuff.o, themainprog.o, and pyramid, using the commands above. Initially, create the targets without any dependencies.

Also add a fourth target called `clean` in your makefile, with the following command:

```
rm -f pyramid stuff.o themainprog.o
```

It is common practice to always have such a `clean` target in makefiles, so that by doing `make clean` all files that have been created previously are removed, thus getting back to a clean state for the code.

Now you should be able to make each target separately, using the following three commands:

```
make stuff.o
make themainprog.o
make pyramid
```

And you should be able to remove the o-files and the executable file by doing `make clean`.

So far, so good. However, we want to be able to build the program using a single command. If we just do "`make pyramid`", we want make to automatically figure out that the targets stuff.o and themainprog.o are needed for that. This can be achieved by specifying dependencies. Add stuff.o themainprog.o as dependencies to the pyramid target, and check that it works. Note that you need to do `make clean` first if you want to test building from the beginning.

We also want the appropriate parts of the code to be recompiled in case some source code file(s) have changed. This is also achieved using dependencies, since make recreates a target if any of its dependencies have changed.

To see how this works, start by looking at one target, e.g. stuff.o. If you do "`make stuff.o`" and the file stuff.o does not exist, make will run the command to create it. However, if the file stuff.o already exists, make will not run the command. This is good since it avoids unnecessary recompilation if nothing has changed, but of course we do want the stuff.o file to be rebuilt if stuff.c has changed. Therefore, stuff.c should be a dependency of the stuff.o target. Do that change in the makefile (add stuff.c as a dependency of stuff.o) and then verify that it works by first doing "`make stuff.o`" once, then changing something in stuff.c and then doing "`make stuff.o`" again. Now make should detect that the dependency has changed, and therefore recreate the target. If you do "`make stuff.o`" again without having changed stuff.c, make should not recreate the target.

Now make sure that all your targets have the appropriate dependencies: stuff.o should have stuff.c as a dependency, and in the same way themainprog.o should have themainprog.c as a dependency.

There is also a header file called stuff.h that is included by both stuff.c and themainprog.c. If something is changed in stuff.h, for example if the value of G there is changed, then both stuff.o and themainprog.o should be recompiled. Add stuff.h as a dependency to those targets and check that this works properly; if you change the value of G in stuff.h and then do "`make pyramid`", then both stuff.o and themainprog.o should be recompiled and the new value of G should be seen when running the program.

When you have checked that your makefile works correctly, part 2 of this assignment is done. Leave your final makefile as the result of part 2, in the part2 directory.

3. Part 3

In this part, you are to create a small shell script. The simplest form of a shell script is just a text file containing some commands that you want to be able to execute as a single command/program.

(In this assignment we will just use this simplest form of shell script, but there is a lot more you can do. If you want to find out more and try doing more advanced things in your scripts, try searching the web for "shell script" and you will find lots of examples and tutorials.)

As a first example, create a text file called `helloscript.sh` with just one line saying "`echo Hej hej`" inside it. To be able to run it as a program, you also need to change file permissions for that file so that it becomes allowed to execute it. This is done using the `chmod` command, like this:

```
chmod +x helloscript.sh
```

After doing `chmod +x` on a file, execute permission has been set for that file which means that it is possible to run it in the same way as an ordinary program would be run:

```
./helloscript.sh
```

When you run your shell script like that you should see the effect of the commands inside it, in this case the "Hej hej" text should be printed.

Now add a few more `echo` lines in your script, and check that when you run it those lines are also executed.

Shell scripts can be very useful for many different purposes. One way of using them that is especially interesting for us in this course is that when we want to test a C program we have written, after compiling the C code we can write a script that automatically runs the code with several different input values. This can make our testing much easier.

To see how this can be done, first compile the pyramid executable from part 2 and copy it into your directory for part 3. The pyramid executable can be run with different input arguments. Try a few different values to see how that works:

```
./pyramid 4
./pyramid 7
./pyramid 17
```

Now create a new shell script `run-pyramid.sh` that runs the pyramid program five times, with the following five input values: 3, 5, 7, 9, 11. When you have checked that your script works correctly, part 3 of this assignment is done. Leave your final scripts as the result of part 3, in the part3 directory.

4. Part 4

In this part you will use the `ssh` and `scp` commands.

You will login to the remote computer `vitsippa.it.uu.se` using `ssh` and then run the `lscpu` command there, storing the resulting text in a file on the remote system. Then you will logout from the remote system, and finally use `scp` to copy the file to your local computer.

Below follows a description of each command you need to use; follow these steps, but use your own username instead of the fake username "`user123`" that is used in the example commands here.

Login to `vitsippa.it.uu.se` using `ssh`:

```
ssh abcd123@vitsippa.it.uu.se
```

After logging in you can verify that you are really there using the `hostname` command, as you learned in Lab 1.

Then run the `lscpu` command:

```
lscpu
```

That should give some information about the processor(s) in the `vitsippa.it.uu.se` computer.

Next, run the `lscpu` command again but this time directing its output to a file:

```
lscpu > lscpu_result.txt
```

Now the file `lscpu_result.txt` should have been created, containing the same text that was previously printed in the terminal window. Check that the file is really there using `ls` and then look at its contents using `cat`:

```
cat lscpu_result.txt
```

Then logout from `vitsippa.it.uu.se` using the `exit` command.

Use `scp` to copy the `lscpu_result.txt` file from `vitsippa.it.uu.se` to your local computer:

```
scp abcd123@vitsippa.it.uu.se:lscpu_result.txt file_from_vitsippa.txt
```

Again, use `ls` and `cat` to check that the local file `file_from_vitsippa.txt` is really there, and that it has the expected contents. When you have checked that, part 4 of this assignment is done. Leave your final `file_from_vitsippa.txt` file as the result of part 4, in the part4 directory.

5. Preparing your submission

To make it easier for your teachers to check your submissions in a systematic way, the submission is required to have a specific form, as described below.

Create a directory called `A1` and put your final `part1`, `part2`, `part3`, and `part4` directories as subdirectories inside the `A1` directory.

No binary files such as object files or executable files should be included.

Then use the `tar` command to create a "tar-ball" package called `A1.tar.gz` containing the `A1` directory with all its contents:

```
tar -czf A1.tar.gz A1
```

When you have created your `A1.tar.gz` file in that way, copy it somewhere else and unpack it there to check that it really has the contents you want.

Before submitting the file in the Student Portal, use the `check-A1.sh` script that was included with the assignment instructions, to verify that your file follows the requested format. To use the `check-A1.sh` script, first set execute permission for it as you learned earlier in this assignment, and then run it when standing in the same directory where you have the `A1.tar.gz` file. If the file has the requested format, the output from the script should say "Congratulations, your A1.tar.gz file seems OK!". If you do not get that result, look carefully at the script output to figure out what went wrong, then fix the problem and try again.

Note that getting the "Congratulations" message from the script does not guarantee that your assignment is fully correct, since the script does not check everything, it mostly just checks that the directories and files exist and are named as requested. When you have submitted the file, your teachers will check your submission more carefully.

Submission

When you are done, upload your final `A1.tar.gz` file in the Student Portal. Note that the uploaded file should have precisely that name, and that you should have checked it using the `check-A1.sh` script before uploading it.