Makefile

Unzip the make-example.zip file provided and enter the example folder. You will see some code and also many files starting with Makefile (do “ls Makefile*”). Each demonstrates a specific aspect of make. Let us learn!

Code

1. The dependency graph for the code is as shown in slides and covered in class.
2. Before we learn make, let us just compile this code
    a. cc -o udp-test *.c
        i. This will fail since the math library is not linked
    b. cc -o udp-test *.c -lm
        i. This will work fine!
3. When you do above, internally the following is happening sequentially
    a. cc -c udp-test.c
    b. cc -c sockutil.c
    c. cc -c timeutil.c
    d. cc -o udp-test *.o -lm
4. Don’t change any file, and do the following
    a. ls -l
        i. Note the time of creation of file udp-test
    b. Wait 2 min
    c. cc -o udp-test *.c -lm; ls -l
        i. Try the above commands and check the time of udp-test, it would have changed.
    d. Even though no file changed it is recompiling everything (of course you asked it to!)

Makefile – Orig

Copy Makefile.orig into a file called Makefile (cp Makefile.orig Makefile). And open the file via some editor (gedit or vi or nano).

1. When you type command “make”, make looks for a file ‘Gnumakefile” or “makefile” or “Makefile”. Hence the copy, where we want our file to be named “Makefile”.
2. The target “all” is a special target that specifies the default target to build when you invoke make without specifying a particular target
    a. It's often used to build the entire project or the main executable
    b. make
        i. When you type make as above, you will see all the commands (4 of them) are run and the executable created!
    c. make all
        i. Make all is the same as make. But since no file changed, you will see something like make: Nothing to be done for 'all'

3. You can also get a specific target to build
    a. make clean
        i. Above will remove all object files (more on this later)
    b. make timeutil.o
        i. Above will only run the command for that specific target!

Makefile - Implicit

Copy Makefile.implicit into a file called Makefile (cp Makefile.implicit Makefile). And open the file via some editor (gedit or vi or nano).

1. You will notice that unlike earlier, there are no commands written under the three ".o" targets. In C, compilation takes a .c file and makes a .o file. Make applies the implicit rule when it sees this combination of file name endings.
    a. make clean; make timeutil.o
        i. Try above, this is no different than the earlier Makefile.orig, same is happening as before
2. Notice also in this makefile, the use of variable OBJ

Makefile - Phony

Copy Makefile.phony into a file called Makefile (cp Makefile.phony Makefile). And open the file via some editor (gedit or vi or nano).

1. This makefile is very different from earlier. Notice the following.
    a. "#" is used to add comments to the makefile
    b. "@echo" prints to stdout whatever is specified in the quotes! The "@" symbol suppresses the command itself from being echoed in the terminal, only displaying the result of the echo command.
    c. There are two targets "clean" and "clean_other". The first target will remove the object files. The other target creates a file called "clean". Notice it has the same name as the first target!
2. make clean_other
    a. This will create a file called clean, if file already exists, it changes its timestamp!
3. make clean
    a. You will expect it will remove the object files, but it will say "clean" is upto date. This is because the file "clean" has not changed. It is not viewing clean as a target.
4. Edit the makefile and remove the # before the .PHONY
    a. .PHONY is a special target that is used to declare certain targets as "phony" targets.
    b. Phony targets are targets that do not represent actual files. Instead, they represent actions that need to be performed during the build process.
    c. By declaring a target as .PHONY, you are telling make that it should not expect a file with that name to be created as a result of the target.
5. make clean

    a. Since you specified .PHONY target, now the clean target will work, you should see the message “cleaning up” .
6. make
    a. Notice there is no “all” target
    b. If you just type make without specifying a target, and there's no explicit “all” target defined in the Makefile, make will attempt to build the first target it encounters in the Makefile

Makefile - Errors

Copy Makefile.errors into a file called Makefile (cp Makefile.errors Makefile). And open the file via some editor (gedit or vi or nano).

1. This makefile is very different from earlier. Notice the following.
    a. .PHONY has two dependencies: all and clean (both are not files)
    b. The commands under target 1/2/3 are dummy i.e. there are no such commands, so if you execute them, bash will say no such file or directory
2. make
    a. Notice a “-” before “./command1”, this will suppress the error and make will move to the next target (target2)
    b. Notice there is no “-” before “./command2”. Make will fail here due to error and not proceed to target3
        i. If there was a “-” before “./command2”, then it would have gone to target3
3. make -k
    a. -k flag will ensure make continues running even in the face of errors. Helpful if you want to see all the errors of Make at once
    b. Notice all three targets were attempted
4. We have so far been copying various Makefile.* to Makefile and exploring them. This is such a pain. How to avoid?
    a. make -f Makefile.errors
        i. If you want to use a file other than `GNUmakefile', `makefile' and `Makefile', you can use the -f option
    b. make -k -f Makefile.errors
        i. Works with two flags!