Make

Zilogic Systems

1. Compiling Programs

- When a program with a single source file is to be compiled, the compiler command can be directly invoked.
- But when there are multiple source files, the process becomes tedious.
- One way to solve the problem is to write a shell script that will invoke all the compilation commands in one shot.
- But this approach does not scale well. When there are thousands of files, as is in the case of the Linux kernel, all files will be compiled everytime the script is invoked. But it is only required to re-compile files that have been modified.
- In the case of the Linux kernel rebuilding the entire kernel might take 10 minutes. But if only those files that are modified are rebuilt it will require just a few seconds.

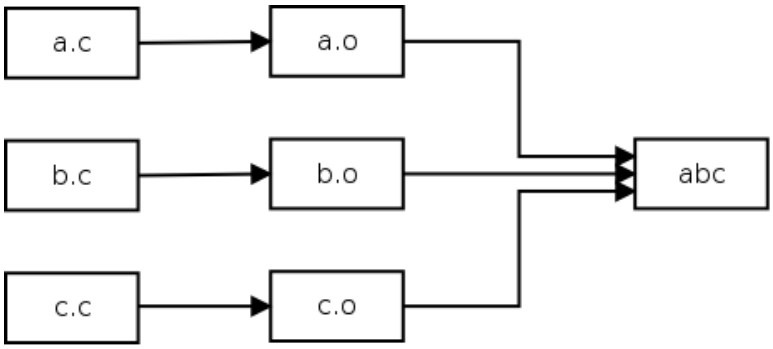
2. `make` Makes it Easy

- `make` is a program that can determine what files have been modified and builds only those files that have been modified.
- `make` acheives this with two inputs a dependency graph and the modification time of the files.
- The dependency graph is specified in a file called the `Makefile`.
- When `make` is invoked it looks for a file called `Makefile` in the current directory.
- The `Makefile` contains a series of rules. Each rule has the following format

```
target: dependency-1 dependency-2 ...
        command-1
        command-2
        ...
```

- The `target` is the file to be created.
- The `dependency-n` are the files that are used to create the `target`.
- The `command-n` are the commands that create `target` from `dependency-n`. Note that the commands are specified by prefixing them with a single tab.

Figure 1. Dependency Graph



- For example, the above dependency graph can be represented using the following `Makefile`.

```
abc: a.o b.o c.o
     gcc -o abc a.o b.o c.o

a.o: a.c
     gcc -c a.c

b.o: b.c
     gcc -c b.c

c.o: c.c
     gcc -c c.c
```

- When make is invoked it reads the `Makefile` and builds the dependency graph.
- Then it tries to bring up-to-date the target corresponding to the first rule, which happens to `abc` in this case.
- Since `abc` depends on `a.o`, `b.o` and `c.o`, `make` first ensures that these are up-to-date.
- So make checks if `a.o` is newer than `a.c`, if not it builds `a.o` using the specified command.
- This is repeated for `a.o`, `b.o` and `c.o`.
- Then if required `abc` is rebuilt from `a.o`, `b.o` and `c.o`.

3. Specifying Targets

- When `make` is invoked without any arguments, the target corresponding to the first rule is brought up-to-date.
- `make` can also be invoked with target to be built as argument.

4. Variables

- A *variable* or *macro* is a name defined in a makefile to represent a string of text, called the variable's *value*.
- These can represent list of files, list of options to be passed to compiler, commands or other parts of makefile or anything else that can be imagined.
- Variable names are case sensitive.
- To substitute a variable's value, use *$* with the name of the variable enclosed in parentheses as shown here:

```
$(variable_name)
```

Variable Substitution.

```
OBJS = a.o b.o c.o
CFLAGS = -Wall

abc: $(OBJS)
     gcc -o abc $(OBJS)

a.o: a.c
     gcc $(CFLAGS) -c a.c
```

```
b.o: b.c
     gcc $(CFLAGS) -c b.c

c.o: c.c
     gcc $(CFLAGS) -c c.c
```

5. Pattern Rules and Special Variables

- In the previous example, the rules for creating `a.o`, `b.o` and `c.o` only differ in their filenames.
- If there are many files, a rule has to be written for each of the files.
- Instead of writing rules for every file to be built, it is posssible to write a generic rule to build a particular type of file.
- For example to build a `.o` file from a `.c` file, the following pattern rule.

```
%.o: %.c
     gcc -c $<
```

- Special variables are available while writing rules.

  $@* The file name of the target of the rule. *$@ is the name of the target file, what so ever, that caused to run the *rule*.

  $< The name of the first prerequisite.

  *$^* The names of all prerequisites with spaces between them.

- So the above example reduces to the following.

```
OBJS = a.o b.o c.o
CFLAGS = -Wall

abc: $(OBJS)
     gcc -o abc $(OBJS)

%.o: %.c
     gcc $(CFLAGS) -c $<
```

6. Phony Targets

- A phony target is not a file name. It is the name of a set of commands which need to be executed when an explicit request is made.

```
clean:
      rm *.o
      rm abc
```

- Since `clean` is not built by the commands. The commands will be always be executed whenever make is required to build the target.
- But if a file called `clean` is accidentally created, since there are no dependencies, the commands will not get executed. To avoid this, the targets that do not really correspond to filenames can be specified as phony as shown below.

```
.PHONY: clean

clean:
```

```
	rm *.o
	rm abc
```

7. Variables from Environment

- Shell environment variables passed to `make` become `make` variables.
- For example to set the `CFLAGS` value, the following command sequence could be used.

```
$ export CFLAGS="-Wall -ggdb"
$ make abc
```

- Another way to set `make` variables from the command line is shown below

```
$ make abc CFLAGS="-Wall -ggdb"
```