Make: a build automation tool

What is the problem?

- The lab examples repository for the CS 253 course has 293 files in 81 folders.
- To build them all would requires us to navigate to 81 folders and compile the files in each folder. . .
  😢
- Imagine a project that has 15 million lines of code in 34,690 files spread over 2386 folders (Linux kernel version 3.11). How would you compile it?!
  😭
- *We need a program to manage the compiling of all the files in our programs!*
- *Make* is such a tool that can automate the build process. E.g. For the Linux kernel, the entire process is driven by *Make*

Demo the make for the 253 example programs

What is build automation?

- Build automation involves automating the process of compiling code into libraries and executables. This can be a very complex process for large projects.
- For large programs, recompiling all the pieces of the program can be very time consuming. If we only recompile the files that have changed, we can save a lot of time.
- But if the program is complex, determining exactly what needs to be recompiled too can be difficult. Build automation also helps with this aspect.
- *Make* is a build automation tool. *Make* and its variants are included with Linux, Mac OS X and MS Windows operating systems.
- Other popular build systems include Apache Maven and Apache Ant. These are used primarily for Java based projects. Gradle is another popular build automation system that builds on Maven and Ant.

What is *Make*? (1)

- *Make* uses a *declarative language* as opposed to procedural languages.
  - We tell *Make* what we want (e.g. a particular class file or executable).
  - We provide a set of rules showing dependencies between files.
  - *Make* uses the rules to get the job done.
- The *Make* program is invoked via the executable named `make`.

What is *Make*? (2)

- *Make* uses a file called `Makefile` (or `makefile`), which contains the set of rules. The recommended name is `Makefile`.
- We recommend using the name `Makefile` because it appears prominently near the beginning of a directory listing, right near other important files such as `README`.
- When we run make, it uses the rules in the `Makefile` to determine what needs to be done.
- *Make* does the minimum amount of work needed to get the job done.
- *Make* can be used to execute an arbitrary set of shell commands and programs so it isn't limited to build automation.

Rules in a Makefile (1)

- A typical rule has the form:

```
target: dependency1 dependency2 ...
        command list
```

- `target` can be the name of a file that needs to be created or a "phony" name that can be used to specify what command to execute.
- The `dependency list` is a space separated list of files that the target depends on in some way. The dependencies are built in the order listed so the order may matter!
- The `command list` is one or more commands needed to accomplish the task of creating the target. The commands can be any shell command or any program in the system.

Rules in a Makefile (2)

- Each command must be indented with a tab.
- Both dependency lists and commands can be continued onto another line by putting a \ at the end of the first line.
- A # is used to start a comment in a makefile.
  - The comment consists of the remainder of the line.

Simplified Doubly-Linked List Example

Dependencies for the simplified doubly-linked list example. For these notes, we assume that all the files for the doubly-linked list are in one folder.

- `SimpleTestList.c` includes `List.h` and `Node.h`
- `UnitTestList.c` includes `List.h`, `Node.h` and `Object.h`
- `List.c` includes `List.h`, `Node.h`
- `Node.c` includes `Node.h`
- `Object.c` includes `Object.h`

Rules for the Simplified Doubly-Linked List

A brute-force approach:

```
UnitTestList: UnitTestList.o List.o Node.o Object.o
    gcc -Wall -g -o UnitTestList UnitTestList.o List.o Node.o Object.o

SimpleTestList: SimpleTestList.o List.o Node.o
    gcc -Wall -g -o SimpleTestList SimpleTestList.o List.o Node.o

List.o: List.c List.h Node.h
    gcc -Wall -g -c List.c

Node.o: Node.c Node.h
    gcc -Wall -g -c Node.c

Object.o: Object.c Object.h
    gcc -Wall -g -c Object.c
```

How `make` works? (1)

- When we type `make` without a target name, it will assume that we mean to build the first real target in the `Makefile`. This is often a phony symbolic target named `all`.
- When we type `make target`, the make utility will look at the rule for `target`
- Make will *recursively* search through the rules for all the dependencies to determine what has been modified and rebuild only those targets

How `make` works? (2)

- If the current version of target is newer than all the files it depends on, make will do nothing.
- If a target file is older than any of the files that it depends on, the command(s) following the rule will be executed

Macros

- Sometimes, we find ourselves using the same sequence of command line options in lots of commands. Use a macro to make it simpler and more robust.
- Define macro as shown below:

```
CC=gcc
CFLAGS=-Wall -g -O
PROGS=SimpleTestList UnitTestList
```

- Then use the macro by typing `$(macroname)`

```
$(CC) $(CFLAGS) -c List.c
```

Substitution Rules

- Often, we will find that our Makefile has many similar commands. We can use patterns to define rules and commands for such cases.
- For example, we could use the rule:

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

- % - any name (the same in all occurrences)
- which says that every .o file depends on the corresponding .c file and can be created from it with the command given below the rule.

Automatic Variables

Most commonly used variables. See https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html for full list.

- `$@` - The name of the current target
- `$<` - The first dependency for the current target
- `$?` - The dependencies that are newer than the current target
- `$^` - All the dependencies for the current target

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

hello: hello.o
	$(CC) $(CFLAGS) $< -o $@
```

Suffix Rules

- A suffix rule identifies suffixes that make should recognize. For example:
  `.SUFFIXES: .o .c`
- Another rule shows how files with suffixes are related:

```
.c.o :
    $(CC) $(CFLAGS) -c $<
```

- Think of this as saying the `.o` file is created from the corresponding `.c` file using the given command.
- Note that the above suffix rule is considered obsolete. The above rule for C files to object files is already built into make.

Phony Targets

- Phony targets are targets that do not correspond to a file

```
all: SimpleTestList UnitTestList

clean:
    rm -force *.o $(PROGS)
```

- Phony targets can be used to create a recursive makefile that can build a project spanning a complex directory structure.

Example: Phony Targets

From `C-examples/linked-lists/Makefile`

```
all: subdirs

subdirs:
    cd buggy-list; make
    cd singly-linked; make
    cd doubly-linked; make

clean:
    cd buggy-list; make clean
    cd singly-linked; make clean
    cd doubly-linked; make clean
```

Simplified Doubly-Linked List Example Makefile

- With macros, suffix rules, and phony targets. Note that the suffix rule shown below is built-in to make, so we can drop the first three lines.

```
.SUFFIXES: .o .c
.c.o :
    $(CC) $(CFLAGS) -c $<
CC=gcc
CFLAGS=-Wall -g -O -I.
LFLAGS=
PROGS=SimpleTestList UnitTestList
LIBOBJS=List.o Node.o
TESTOBJS=Object.o

all: $(PROGS) dox
SimpleTestList: SimpleTestList.o $(LIBOBJS)
    $(CC) $(CFLAGS) -o $@ $^ $(LFLAGS)

UnitTestList: UnitTestList.o $(LIBOBJS) $(TESTOBJS)
    $(CC) $(CFLAGS) -o $@ $^ $(LFLAGS)

dox:
    echo "Generating documentation using doxygen..."
    doxygen doxygen-config > doxygen.log
    echo "Use konqueror docs/html/index.html to see docs (or another browser)
    "

clean:
    /bin/rm -f $(PROGS) *.o a.out
    /bin/rm -fr docs doxygen.log
```

Taking the drudgery out of dependencies

- Dependencies for a `.o` file should include all the user written header files that it includes. The previous Makefile didn't do that....
- For a big project, getting all of these right can take some time
- The `gcc` command has an option `-MMD` that tells it to compute the dependencies.
- These are stored in a file with the suffix `.d`
- Include the `.d` files into the `Makefile` using `-include *.d`

The Final Makefile for the Simplified Doubly-Linked List

```
CC=gcc
CFLAGS=-Wall -g -O -I. -MMD
LFLAGS=
PROGS=SimpleTestList UnitTestList
LIBOBJS=List.o Node.o
TESTOBJS=Object.o

all: $(PROGS) dox
SimpleTestList: SimpleTestList.o $(LIBOBJS)
    $(CC) $(CFLAGS) -o $@ $^ $(LFLAGS)

UnitTestList: UnitTestList.o $(LIBOBJS) $(TESTOBJS)
    $(CC) $(CFLAGS) -o $@ $^ $(LFLAGS)

-include *.d

dox:
    echo "Generating documentation using doxygen..."
    doxygen doxygen-config >& doxygen.log
    echo "Use konqueror docs/html/index.html to see docs (or another browser)
    "

clean:
    /bin/rm -f $(PROGS) *.o a.out
    /bin/rm -fr docs doxygen.log
```

The Complete Doubly-Linked List Makefiles

- Directory listing for the project (in a clean state)

```
[amit@localhost doubly-linked(master)]$ ls -R
.:
doxygen-config  include  lib  libsrc  Makefile  README.md
run.sh  testsuite
./include:
./lib:
./libsrc:
List.c  List.h  Makefile  Node.c  Node.h
./testsuite:
Makefile  Object.c  Object.h  RandomTestList.c
SimpleTestList.c  UnitTestList.c
```

- The Makefile at the top-level invokes the Makefile in libsrc subfolder to build and install the list library.
- Then it invokes the Makefile in the testsuite folder to build the test programs using the list library.

Additional features

- **Multiple rules for a target**.
  - If there is more that one rule for a given target, make will combine them.
  - The rules can be specified in any order in the `Makefile`
- **Parallel make**. Use the `-j` option to have make generate your project using multiple CPUs to speed up the building process! Make will build multiple dependencies for a rule in parallel. Note that this does require you to check that the various dependencies can be built simultaneously.
- Try the following commands in sequence on the class examples on a machine in the lab (or any machine with at least 4 cores):

```
time make
make clean
time make -j 4
```

- Did it build faster? If not, why not?
- What happens if we use `make -j` without the number of threads after the `-j` option?

References

- Wikipedia entry on Make:
  `http://en.wikipedia.org/wiki/Make_(software)`
- GNU Make homepage:
  `https://www.gnu.org/software/make/`
- Managing projects with GNU Make.
  `http://www.wanderinghorse.net/computing/make/`
  (downloadable book)