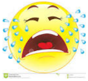
Make: a build automation tool

What is the problem?

- The lab examples repository for the CS 253 course has 228 files in 54 folders.
- To build them all would requires us to navigate to 54 folders and compile the files in each folder...

- Imagine a project 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.

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`. Why?
- 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.

Doubly-Linked List Example

Dependencies for the doubly-linked list

- `SimpleTestList.c` includes `List.h`, `Node.h`, `Job.h`, and `common.h`
- `List.c` includes `List.h`, `Node.h`, `Job.h`, and `common.h`
- `Node.c` includes `Node.h`, `Job.h`, and `common.h`
- `Job.c` includes `Job.h` and `common.h`

Rules for Doubly-Linked List

A brute-force approach:

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

SimpleTestList.o: SimpleTestList.c List.h Node.h Job.h common.h
    gcc -Wall -g -c SimpleTestList.c

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

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

Job.o: Job.c Job.h common.h
    gcc -Wall -g -c Job.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 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 RandomTestList 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 $<
```

- Which says that every `.o` file depends on the corresponding `.c` file and can be created from it with the command below the rule.

Substitution Rules - Internal macros

- % - any name (the same in all occurrences)
- $@ - 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 the above suffix 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 RandomTestList

  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-example/doublyLinkedList/Makefile`

```
all: subdirs

subdirs:
    cd bad; make
    cd almost-generic; make
    cd generic-with-library; make
    cd generic; make

clean:
    cd bad; make clean
    cd almost-generic; make clean
    cd generic-with-library; make clean
    cd generic; make clean
```

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 RandomTestList
OBJECTS=List.o Node.o Job.o

all: $(PROGS) dox
SimpleTestList: SimpleTestList.o $(OBJECTS)
    $(CC) $(CFLAGS) -o $@ $^ $(LFLAGS)
RandomTestList: RandomTestList.o $(OBJECTS)
    $(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 Doubly-Linked List

```
CC=gcc
CFLAGS=-Wall -g -O -I. -MMD
LFLAGS=
PROGS=SimpleTestList UnitTestList RandomTestList
OBJECTS=List.o Node.o Job.o

all: $(PROGS)
SimpleTestList: SimpleTestList.o $(OBJECTS)
    $(CC) $(CFLAGS) -o $@ $^ $(LFLAGS)
UnitTestList: UnitTestList.o $(OBJECTS)
    $(CC) $(CFLAGS) -o $@ $^ $(LFLAGS)
RandomTestList: RandomTestList.o $(OBJECTS)
    $(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 any other browser)"
clean:
    /bin/rm -f $(PROGS) *.o a.out
    /bin/rm -fr docs doxygen.log
```

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 simulatenously.
- 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?

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)