Today’s Lecture

- Brief review of make
- Explore make “macros” in more detail
  - Note: when you see “macro” think “variable”
- Brooks’ Corner: The Mythical Man-Month

- but first…a quick look at Ant (a build management tool for Java programs)

Unix Build Management

- In Unix environments, a common build management tool is “make”
  - Make provides very powerful capabilities via three types of specification styles
    - declarative
    - imperative
    - relational
  - These styles are combined into one specification: “the make file”

Make Specification Language

- Hybrid Declarative/Imperative/Relational
  - Dependencies are Relational
    - Make specifies dependencies between artifacts
  - Rules are Declarative
    - Make specifies rules for creating new artifacts
  - Actions are Imperative
    - Make specifies actions to carry out rules
Example “Makefile”

<table>
<thead>
<tr>
<th>Target</th>
<th>Dependencies</th>
<th>Actions</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: T2 T3 T4</td>
<td>A1 A2 A3</td>
<td></td>
<td>T1: T2 T3 T4</td>
</tr>
<tr>
<td>T2: T5 T6</td>
<td>A4</td>
<td></td>
<td>T2: T5 T6</td>
</tr>
<tr>
<td>T3: T5 T7</td>
<td>A5 A6</td>
<td></td>
<td>T3: T5 T7</td>
</tr>
</tbody>
</table>

If a dependency changes, a rule’s actions are executed to (re)create a rule’s target

Make “Macros” - think “Variables”

- Make has variables known as “macros”
  - They are similar to shell variables with a few differences
    - Macros hold a string value
    - Macros are defined using an equal sign
      \[ \text{INSTALLDIR} = /home/faculty/kena/tmp/ \]
    - And is used by preceding its name with a dollar sign
      \[ $(\text{INSTALLDIR})/\text{program} : \text{program} \]
      \[ \text{cp} \text{ program} $(\text{INSTALLDIR})/\text{program} : \text{program} \]
    - The parentheses are required, otherwise make assumes that a macro name is just one letter long
      \[ $(\text{INSTALLDIR}) \]
      \[ $(\text{INSTALLDIR}) \]
- \$INSTALLDIR is interpreted by make as $(/\text{INSTALLDIR})

Macro Substitution

- Make variables perform strict textual replacement so the following two rules are equivalent
  - (Do not do this in practice!):
    \[
    \text{program: output.o} \\
    \text{g++ output.o -o program} \\
    \text{FOO = o} \\
    \text{pr$(\text{FOO})\text{gram: $(\text{FOO})output.$(\text{FOO})} \\
    \text{g++ $(\text{FOO})output.$(\text{FOO}) -$$(\text{FOO}) pr$(\text{FOO})\text{gram} \]

Using a ‘$’ sign

- Since the dollar sign has special meaning…
  - it indicates the use of a macro
  - …you need to “escape” it with a 2nd dollar sign, if you want it passed to the shell as part of an action
    - Note: make strips one of the dollar signs before invoking a shell to process the action
  - Example: ‘chapter$’ is passed to \text{egrep} below
    \[
    \text{TableOfContents: book.txt} \\
    \text{egrep chapter$ book.txt > TableOfContents} \]
Increased Abstraction

- Macros increase the level of abstraction in a Makefile program: main.o input.o output.o
  
g++ main.o input.o output.o -o program

- is equivalent to
  
EXECUTABLE = program
OBJECTS = main.o input.o output.o

$(EXECUTABLE): $(OBJECTS)
  
g++ $(OBJECTS) -o $(EXECUTABLE)

- They can also save keystrokes

Increased Abstraction, cont.

- Why is this increase in abstraction important?
  - What benefit does abstraction typically provide?

Definition of Abstraction

- Identify the important aspect of a phenomenon and ignore the details

Increased Abstraction, cont.

- Allows the user of an abstraction to be independent of the hidden details
  - This allows the details to change without a user knowing about it (or caring)

- In makefiles, abstraction lets rules be defined that can be applied to many different situations

  $(EXECUTABLE): $(OBJECTS)
  
g++ $(OBJECTS) -o $(EXECUTABLE)

- The above rule can be applied to almost any C++ or C program

Definition and Use of Make Macros

- A shell script is executed from top to bottom. As such, a shell variable cannot be used before it is defined.

- Makefiles, on the other hand, are not executed top to bottom. Execution follows dependencies which can be anywhere in the file
  - As such, there is no concept of one rule coming before or after another rule
  - Therefore, all rules and macros are read entirely before the make algorithm is executed
Definition and Use, continued

- Shell Variables
  ```bash
  %echo $var
  %set var = hello
  ```
- In response to the first statement, the shell complains “undefined variable”
- Make Macros
  ```bash
  make:
    echo $(VAR)
  VAR = hello
  ```
- Running `make` on the above makefile produces
  `echo hello` `hello`

Advanced Macro Use

```bash
BASEDIR = $(HOME)/csci3308
SRCDIR = $(BASEDIR)/src/function
ARCHDIR = $(BASEDIR)/arch/$(ARCH)
BUILDDIR = $(ARCHDIR)/build/function
BINDIR = $(ARCHDIR)/bin
MANDIR = $(ARCHDIR)/man
SOURCE = function.cpp
OBJECT = function.o
EXEC = function

$(BUILDDIR)/$(OBJECT): $(SRCDIR)/$(SOURCE)
g++ -c $(SRCDIR)/$(SOURCE) -o $(BUILDDIR)/$(OBJECT)
$(BINDIR)/$(EXEC): $(BUILDDIR)/$(OBJECT)
g++ $(BUILDDIR)/$(OBJECT) -o $(BINDIR)/$(EXEC)
```

Brooks’ Corner: The Mythical Man-Month (Chapter 2)

- Cost does indeed vary as the product of the number of workers and the number of months
  - Progress does not!
  - The unit of the man-month implies that workers and months are interchangeable
  - However, this is only true when a task can be partitioned among many workers with no communication among them!

The Man-Month, continued

- When a task is sequential, more effort has no effect on the schedule
  - “The bearing of a child takes nine months, no matter how many women are assigned!”
  - Many tasks in software engineering have sequential constraints!
Most tasks require communication among workers:
- Communication consists of training and sharing information (intercommunication).
- Training affects effort at worst linearly.
- Intercommunication adds \( \frac{n(n-1)}{2} \) to effort if each worker must communicate with every other worker.

**Intercommunication Effort**

<table>
<thead>
<tr>
<th>Workers</th>
<th>Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1 path</td>
</tr>
<tr>
<td>3</td>
<td>3 paths</td>
</tr>
<tr>
<td>4</td>
<td>6 paths</td>
</tr>
<tr>
<td>5</td>
<td>10 paths</td>
</tr>
<tr>
<td>6</td>
<td>15 paths</td>
</tr>
<tr>
<td>7</td>
<td>21 paths</td>
</tr>
</tbody>
</table>

**Comparison Graphs**

“Adding more people then lengthens, not shortens, the schedule!”
-- (A paraphrase of) Brooks’ Law

**Scheduling**

- Brook’s rule of thumb:
  - 1/3 planning
  - 1/6 coding
  - 1/4 component test
  - 1/4 system test
  - More time devoted to planning, half to testing!
- In looking at other projects, Brooks found that few planned for 50% testing, but most spent 50% of their time testing!
  - Many of these projects were on schedule until testing began!