CSE 15L: Software Tools and Techniques Laboratory

Winter 2021 - http://ieng6.ucsd.edu/~cs15x

Instructors: Gary Gillespie Keith Muller

Class sessions will be recorded and made available to students asynchronously.

Schedule

Today

1. More On Shell Scripts
2. Files: Timestamps and Size
3. Introduction to Build Automation
4. Using **`make`** and Makefiles

Quote Characters

Three different quote characters with different behavior:

" double quote, weak quote

If a string is enclosed in " " the references to variables (i.e *$variable. - later lecture*) are replaced by their values

- back-quote and escape \ characters are treated specially.

' single quote, strong quote

Everything inside single quotes are taken literally; nothing is treated as special.

` back quote (back tick) – alternative is $(cmd)

Enclosed string is treated as a command and the shell attempts to execute it. If the execution is successful, the primary output from the command replaces the string.

Slide ref: *its2.unc.edu/divisions/rc/training/scientific/short.../**Shell_Scripting.ppt*** byShubin Liu, Ph.D.

Controlling Expansion: Quoting

- *word splitting* is where the shell removes extra whitespace from a command's list of arguments
  - word splitting looks for the presence of spaces, tabs, and newlines (line feed characters) and treats them as *delimiters* between words
  - unquoted spaces, tabs, and newlines are not considered to be part of the text. They serve only as separators

```
$ echo this is a          test
this is a test
```

- *double quotes* all the special characters used by the shell lose their special meaning and are treated as ordinary characters.
  - The exceptions are $ (dollar sign), \ (backslash), and ` (backtick).

```
$ echo "this is a          test"
this is a          test
```

Command Substitution

$(cmd args) allows us to use the output of a command as an expansion

Example

- **which** returns the pathnames of the files (or links) which would be executed in the current environment, had its arguments were used as commands in a shell

```
$ which cp
/usr/bin/cp
$ ls -ls $(which cp)
112 -rwxr-xr-x 1 root root 112780 Feb 28  2019 /usr/bin/cp
```

Use and Commenting

- Lines starting with # are comments except the very first line which starts with a #! (the #! if often referred to as the ***shebang***) that tells the OS the absolute path to desired shell for interpreting the shell script
  #!/usr/bin/bash
- On any line, characters following unquoted # are comments and ignored
- Comments are used to:
  - Identify who wrote it and when
  - Identify input variables
  - Make code easy to read
  - Explain complex code sections
  - Version control tracking
  - Record modifications

Slide ref: *its2.unc.edu/divisions/rc/training/scientific/short.../**Shell_Scripting.ppt*** byShubin Liu, Ph.D.

User Input During Shell Script Execution

User input from stdin (terminal or file) can be captured using the **read** command

```
$ cat exampleread.sh
#!/usr/bin/bash
echo "Please enter three filenames:"
read  filea fileb filec
echo "These files are used:$filea $fileb $filec"
```

Each **read** statement reads an entire line (terminated by \n).

In the above example, if there are less than 3 items in the response the trailing variables will be set to blank ' '.

```
$ ./exampleread.sh
Please enter three filenames:
a b c
These files are used:a b c
```


Slide ref: *its2.unc.edu/divisions/rc/training/scientific/short.../**Shell_Scripting.ppt*** byShubin Liu, Ph.D.

Debugging shell scripts

Generous use of the **`echo`** command will help.

Run script with the **`–x`** parameter.

e.g. **`$ bash –x ./myscript`**

or **`$ set –o xtrace`** # before running the script.

These options can be added to the first line of the script where the shell is defined.

e.g. **`#!/usr/bin/bash -xv`**

Slide ref: *its2.unc.edu/divisions/rc/training/scientific/short.../**Shell_Scripting.ppt*** byShubin Liu, Ph.D.

Access Rights to Files and Directories

- *file mode*, describe the read, write, and execute permissions for the file's owner, the group owner (many users in a group), and everybody else (world). Groups lists groups you are a member of

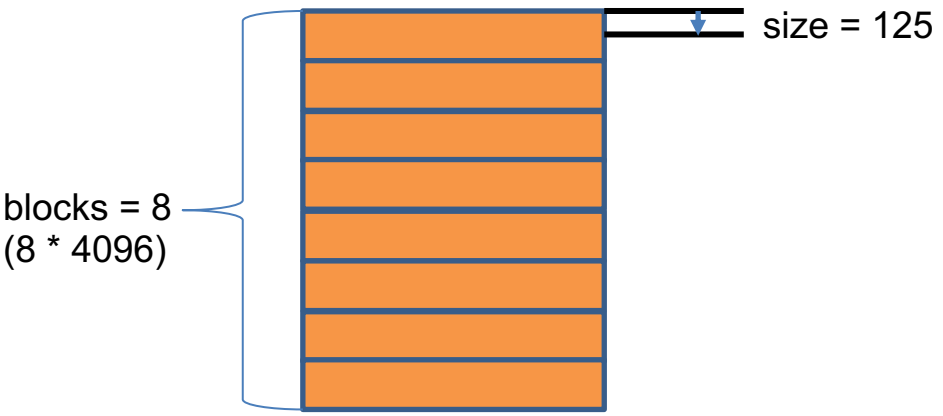
```
$ groups
kmuller adm sudo audio video plugdev games users netdev lpadmin gpio i2c spi
```

Special	Owner	Group	World (other)
suid, segid, restricted_deletion	rwx	rwx	rwx

Attribute	Files	Directories
r	Allows a file to be opened and read.	Allows a directory's contents to be listed if the execute attribute is also set.
w	Allows a file to be written to or truncated; however, this attribute does not allow files to be renamed or deleted. The ability to delete or rename files is determined by directory attributes.	Allows files within a directory to be created, deleted, and renamed if the execute attribute is also set.
x	Allows a file to be treated as a program and executed. Program files written in scripting languages must also be set as readable to be executed.	Allows a directory to be entered, e.g., `cd` *`directory`*.

File and Directory Attributes: Size & Time

```
$ cat main.c
#include<stdlib.h>
#include<stdio.h>
char mesg[]="Hello World\n";
int
main(void)
{
    printf(mesg);
    return EXIT_SUCCESS;
}
$ stat !$
stat main.c
  File: main.c
  Size: 125       Blocks: 8          IO Block: 4096   regular file
Device: 802h/2050d Inode: 644924      Links: 1
Access: (0644/-rw-r--r--) Uid: (1001/ kmuller)  Gid: (1001/ kmuller)
Access: 2021-01-15 08:53:34.973801688 -0800
Modify: 2020-06-03 14:33:24.000000000 -0700
Change: 2021-01-15 08:53:34.973801688 -0800
```



!$ is a shortcut for last arg of previous command

Last time file data accessed read/write

Last time file data was modified (written)

Last time file data or metadata was changed

File and Directory Attributes: Timestamps

```
$ touch main.c
$ !s
stat main.c
  File: main.c
  Size: 125       Blocks: 8          IO Block: 4096   regular file
Device: 802h/2050d Inode: 644924      Links: 1
Access: (0644/-rw-r--r--)  Uid: (1001/ kmuller)   Gid: (1001/ kmuller)
Access: 2021-01-27 11:15:50.228359591 -0800
Modify: 2021-01-27 11:15:50.228359591 -0800
Change: 2021-01-27 11:15:50.228359591 -0800

$ cat main.c > /dev/null
$ !st
stat main.c
  File: main.c
  Size: 125       Blocks: 8          IO Block: 4096   regular file
Device: 802h/2050d Inode: 644924      Links: 1
Access: (0644/-rw-r--r--)  Uid: (1001/ kmuller)   Gid: (1001/ kmuller)
Access: 2021-01-27 11:15:50.228359591 -0800
Modify: 2021-01-27 11:15:50.228359591 -0800
Change: 2021-01-27 11:15:50.228359591 -0800
```

Last time file data was accessed read/write

Last time file data was modified (written)

Last time file data or metadata was changed

Last time file data was accessed read/write (NOATIME Disabled on SSD's performance optimization)

File and Directory Attributes: Timestamps

```
$ touch -a main.c
$ !st
stat main.c
  File: main.c
  Size: 125       Blocks: 8          IO Block: 4096   regular file
Device: 802h/2050d Inode: 644924      Links: 1
Access: (0644/-rw-r--r--)  Uid: (1001/ kmuller)   Gid: (1001/ kmuller)
Access: 2021-01-27 11:25:03.093901779 -0800
Modify: 2021-01-27 11:15:50.228359591 -0800
Change: 2021-01-27 11:25:03.093901779 -0800
```

Last time file data was accessed read/write

Last time file data was modified (written)

Last time file data or metadata was changed

Make

- **`make`** is a program for controlling what gets (re)compiled/recreated and how
  - Many other such programs exist (*e.g.* `ant`, `maven`, IDE "projects")
- **`make`** has tons of complex features, but only two basic ideas:

  1) Scripts for executing commands
  2) Dependencies for avoiding unnecessary work
- In industry, Programmers spend a lot of time "building"
  - Creating programs from source code
  - Both programs that they write, and other people write
- Programmers like to automate repetitive tasks
  - Repetitive: **`gcc -Wall -g -o widget foo.c bar.c baz.c`**
    - Retype this every time: 😭
    - Use up-arrow or history: 😐
    - Have an alias or bash script: 🙂
    - Have a Makefile: 😊

Recompilation Management

- The "theory" behind avoiding unnecessary compilation/processing is a ***modification time-based*** *dependency DAG*

  (**D**irected, **A**cyclic **G**raph)

- To create a target t,
  - you need sources $s_1, s_2, \ldots, s_n$
  - and command(s) c that directly or indirectly uses the sources to create the targets
- It t is newer than every source (based on file-modification times by the OS)
  - assume there is no reason to rebuild it
- Recursive building: if some source s_i is itself a target for some other sources, see if it needs to be rebuilt…

Theory Applied to C

```
#include "foo.h"
char *
foo(void)
{
}
```

file: foo.c

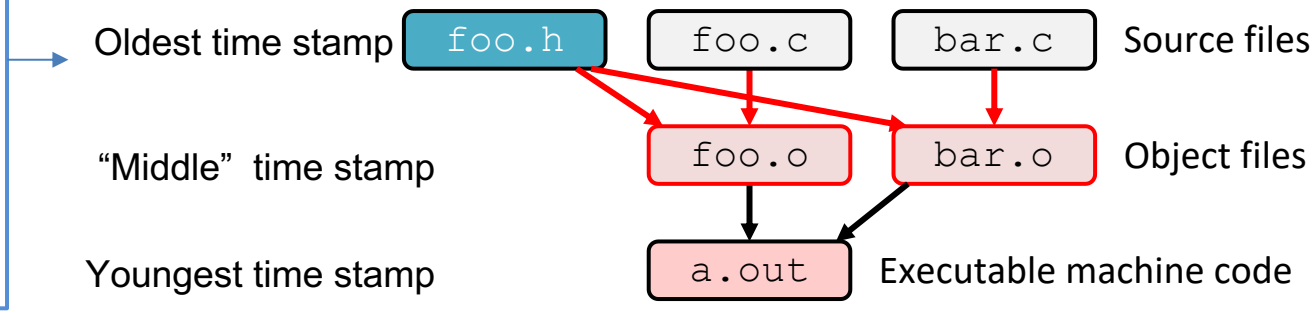
```
#define XXX 7
```

file: foo.h

```
#include "foo.h"
int
bar(void)
{
    char *x = foo();
}
```

file: bar.c

this is the DAG use file change times to determine which files are up to date



- Compiling a `.c` creates a `.o`
  - `gcc -c foo.c` creates foo.o
  - the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)
- Creating an executable ("linking") depends on `.o` files
  - `gcc foo.o bar.o` creates a.out
- If one `.c` file changes, all that really needs to be done is top recreate one `.o` file, maybe a library, and re-link
- If a `.h` file changes, may need to rebuild more

make and Makefiles

- One simple but powerful scripting framework is **make**
- When **make** is run, it looks for a file named **Makefile** in the current working directory
- The **Makefile** contains rules, which tell **make** what to do
- Yof can also use –f to specify a different makefile

```
make -f makefile
```

`make` and simple Makefile rules

- **`make`** interprets a simple Makefile rule as a script with the given target
- Each rule has a target
  - The target refers to a filename in the current working directory!
- A Makefile can have multiple rules, each with a different target
- Running **`make name`**$_i$ will execute the actions in the rule that has target **`name`**$_i$
- Running **`make`** will execute the first rule by default

General Makefile Rule Structure

- Makefile Rules have the following form:

```
target : dependency1 … dependencyM
        action1
        …
        actionN
```

Use TAB!
Not space

Each line of the script can use standard Unix commands

Aside: VIM and tab expansion

```
$ cat makefile
hello: hwl.o
    gcc hwl.o -o hello

hwl.o: hwl.s
    gcc -c hwl.s
clean:
    rm -f hwl.o hello
$ cat Makefile
hello: hwl.o
    gcc hwl.o -o hello

hwl.o: hwl.s
    gcc -c hwl.s
clean:
    rm -f hwl.o hello
$ make -f Makefile
make: 'hello' is up to date.
$ make -f makefile
makefile:3: *** missing separator.  Stop.
$ od -c makefile
0000000  \n   h   e   l   l   o   :       h   w   l   .   o  \n
0000020          g   c   c       h   w   l   .   o       -   o       h
0000040   e   l   l   o  \n  \n   h   w   l   .   o   :       h   w   l
0000060   .   s  \n                   g   c   c       -   c       h   w
0000100   l   .   s  \n   c   l   e   a   n   :  \n                   r
0000120   m       -   f       h   w   l   .   o       h   e   l   l   o
0000140  \n
```

```
in file .vimrc
set shiftwidth=4
set tabstop=4
set noexpandtab
```

General Makefile Rule Structure

```
target : dependency1 … dependencyM
        action1
        …
        actionN
```

- Dependencies may also be targets in the Makefile
- If any dependency is older, **`make`** will try to re-make it FIRST!
- This creates chains of dependence: a file can depend on some files, which depend on other files, etc. **`make`** can figure all this out!

Rule Interpretation

```
target : dependency1 … dependencyM
      action1
      …
      actionN
```

"to make the **target**,

first make all its **dependencies**,

then perform all the **actions**"

Makefile timestamps and dependencies

- The dependencies in a rule can be the names of other targets in the Makefile
- In that case, a rule like

```
target0 : target1 … targetM
        action1
        …
        actionN
```

will execute the actions for **target$_1$** through **target$_M$**, and then the actions for **target$_0$**

- The actions will be performed if any of the dependencies are newer than the target.

Make Basics

- A Makefile contains a bunch of triples:

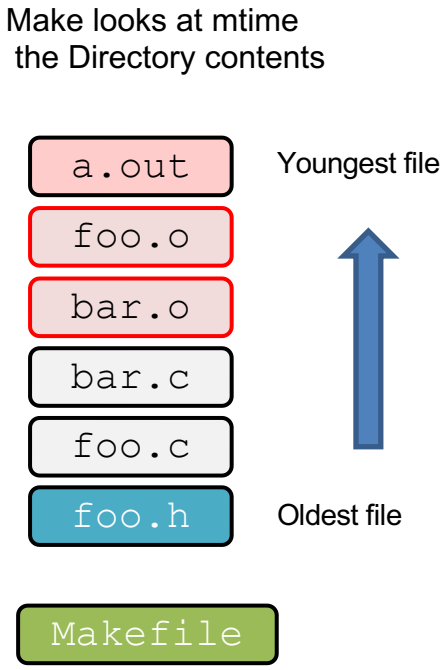
```
target: source_1, … source_N
← Tab → command(s)
```

1. **Target compared to sources by file modification time**
2. **When any of source_1 to source_N is younger (newer file modification time) than target then run the command**
3. **Running command must update/create/"touch" the target**
   - Colon after target is *required*
   - Command lines must start with a **TAB**, NOT SPACES
   - Multiple commands for same target are executed *in order*
     - Can split commands over multiple lines by ending lines with '\'

- Example:

```
bar.o: bar.c foo.h bar.h
        gcc -Wall -c bar.c
```

do not forget the tab!!!!



Using Make

```
$ make -f <makefileName> target
```

- Defaults:
  - If no `-f` specified, (this is what I do) use a file named `Makefile`
  - If `target` is NOT specified, the target will be the first target in the Makefile
  - Will interpret commands in your default shell
    - Set `SHELL` variable in Makefile to use a different shell
- Target execution:
- Check each source in the source list:
  - If the source is a target in the Makefile, then process it recursively
  - If some source does not exist, then error
  - If any source is newer than the target (or target does not exist), run `command` (presumably to update the target)

Simple Makefile example

"to make the **target**, first make all its **dependencies**, then perform all the **actions**"

What will happen if we **`make all`** for the following Makefile?

What if we **`make all`** again?

```
all: target1 target2
	echo "All done!"

target1:
	echo "Line 1 printed"

target2:
	echo "Line 2 printed"
	touch target2
```

Simple Makefile Run

```
$ make                    (or make all)
echo "Line 1 printed"
Line 1 printed
echo "Line 2 printed"
Line 2 printed
touch target2
echo "All done!"
All done!
$ make
echo "Line 1 printed"
Line 1 printed
echo "All done!"
All done!
```

The .PHONY target

- A target that is always out of date!

```
clean:
	rm -rf *.class
```

be real careful with `rm -r`

What if there was a file called clean in the same directory?

```
.PHONY: clean
clean:
	rm -rf *.class
```


More examples at: http://makefiletutorial.com/

Command Execution in `Make`

- Each command line is executed in a separate shell
- Use `;` (semicolon) between commands to execute sequentially!
- If lines are too long, use `\` (backslash) to continue on the next line

Command Execution Example

```
all:
	cd ..
	# The cd above does not affect line
below
	echo `pwd`

	# This cd command affects the next
	# because they are on the same line
	cd ..;echo `pwd`

	# Same as above
	cd ..; \
	echo `pwd`
```

Make in Java Development

- Example: in basic Java development, you could have these rules in a Makefile:

```
Prog.class: Prog.java
   javac Prog.java
run: Prog.class
     java Prog
```

- Now: running **`make run`** will compile Prog.java if it doesn't exist or is newer than Prog.class, and execute the program

Makefile macros

- Makefiles can contain macros, which act like variables.
- For example, if you have a lot of files in your java project, define a macro like:

```
CLASSES = A.class B.class X.class Y.class
```

- And then write the rule:

```
all: $(CLASSES)
```

- Now with the suffix rule shown before, running **`make all`** will compile all the .java files into .class files

make macros

Example (Makefile contents) :

```
CC = gcc
CFLAGS = -Wall -g
bar.o: bar.c foo.h bar.h
        $(CC) $(CFLAGS) -c bar.c
```

- Easy to change things (especially in multiple commands)
- Can also specify on the command line (over-ride for OSX for example):
  (*e.g.* `make foo.o CC=clang CFLAGS=-g`)

```
OBJS = foo.o bar.o baz.o
widget: $(OBJS)
        gcc -Wall $(OBJS)-o widget
clean:
        rm -f $(OBJS) widget
```

Suffix Directive

- For every file X.java in your Java project, you could write a rule

```
X.class: X.java
	javac X.java
```

- But if you have a lot of such files, it would be tedious to write a rule for each of them
- By using a **suffix directive**, you can write just one rule that handles all the files at once

Suffix Directives

- In the Makefile, write the suffix directive:

```
.SUFFIXES: .java .class
```

- And then write the suffix rule:

```
.java.class:
    javac $<
```

- The **$<** symbol means: the dependency, whatever it is

  (Like any rule, the action line must start with a tab)

- Now any .class target file will be made from the corresponding .java file
- *Note:* `Make` *was originally made with C in mind and has built-in implicit rules similar to these for handling .c and .o files!*

Automatic Variables

When writing Makefiles, you will want to reference targets, dependency names, etc.

Use **`make`**'s automatic variables!

- **`$@`** filename of the target
- **`$%`** target member name
- **`$<`** first dependency
- **`$^`** all dependencies
- **`$?`** all dependencies newer than target

Longer list: https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html