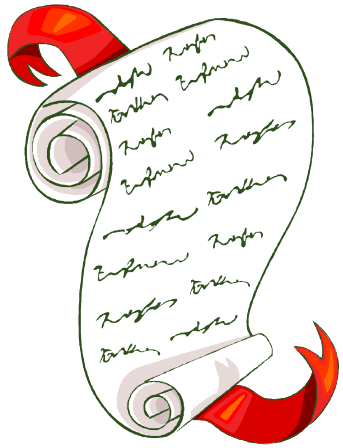
Video Lecture # 04
UNIX make Utility

Course: SYSTEM PROGRAMMING

Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

Today's Agenda



- Introduction to UNIX `make` utility
- Structure of UNIX `makefile`
- How `make` utility work (Examples 1-2)
- Multiple targets in a `makefile` (Example 3)
- Multiple `makefiles` in a Project (Example 4)
- Use of macros in a `makefile` (Example 5)

Make Utility-Introduction

1. Imagine you write a program and divide it into hundred `.c` files and some header files
2. To make the executable you need to compile those hundred source files to create hundred relocatable object files and then you need to link those object files to final executable
3. What happens if we make changes to one of these files:
   (a) Recompile all the files and then link all of them
   (b) Recompile only the file which has changed and then link
   - What if instead of .c file a .h file has changed
   - Solution: Recompile only those .c files that include this header file and then link
4. UNIX `make` utility is a powerful tool that allows you to manage compilation of multiple modules into an executable
5. It reads a specification file called "`makefile`" or "`Makefile`", that describes how the modules of a s/w system depend on each other. If you want to use a non-standard name you can specify that name to make using -f option
6. Make utility uses this dependency specification in the `makefile` and the time when various components were modified, in order to minimize the amount of recompilation

Structure of Makefile

Name of the files on which the target depends (.c and .h files)

Name of the executable to be build

```
target :  dependency1 dependancy2 ... dependency n
<tab>     command
```

Shell command to create the target from dependencies

1. This is one **dependency rule** in a `makefile`
2. A `makefile` may have several such rules. Every make rule describes the dependency relationship
3. Advantages of `make` utility:
   (a) Makes management of large s/w projects with multiple source files easy
   (b) No need to recompile a source file that has not been modified, only those files that have been changed are recompiled, others are simply relinked

Examples 1-2

Options to make

`make[options]`

There are several options to make. For details refer to man page. The three most commonly used are:

-f	By default make looks for a file “makefile” in the current directory. If doesn't exist, it looks for “Makefile”. To tell make to use a different file, user -f option followed by filename
-n	To tell make to print out what it would have done w/o actually doing it
-k	Tells make to keep going when an error is found, rather than stopping as soon as the first problem is detected. You can use this to find out in one go which source files fail to compile

Multiple Targets in a Makefile

- A `makefile` can have multiple targets. We can call a make file with the name of a particular target
- To tell `make` to build a particular target, you can pass the target name to make as parameter (By default, `make` will try to make the first target listed in `makefile`)
- Many programmers specify `all` as the first target in their `makefile` and then list the other targets as being dependencies for all
- A phony target is a target without dependency list. Some important phony targets are `all`, `clean`, `install`

```
clean:
    -@rm    -f    *.o
```

- If there is no .o file in the current working directory, `make` will return an error. If we want make to ignore error while executing a command we proceed the command with a hyphen as done above. Moreover, make print the command to stdout before executing. If we want to tell make not to print the command to stdout before executing we use @ character

Example 3

Multiple Makefiles in a Project

- Project source divided in multiple directories
- Different developers involved
- Multiple makefiles
- Top level makefile use include directive
- **Include Directive:** Tells make to suspend reading the current makefile and read one or more other makefiles before continuing.

```
include ./d2/makefile  ./d3/makefile
```

Example 4

Use of Macros in a Makefile

- A `Makefile` allows us to use macros or variables, so that we can write it in a more generalized form. Variables allow a text string to be defined once and substituted in multiple places later
- We can define macros/variables in a `makefile` as:

```
MACRONAME=value
```

- We can access the macros as `$(MACRONAME)`
- **Example:** We can use a macro to give options to the compiler, e.g., while an application is being developed, it will be compiled with no optimization but with debugging information included. So we declare a macro CFLAGS

```
CFLAGS = -std=c11 -O0 -ggdb -Wall
```

and later can use it with all compilation commands like

```
gcc -c file.c $(CFLAGS)
```

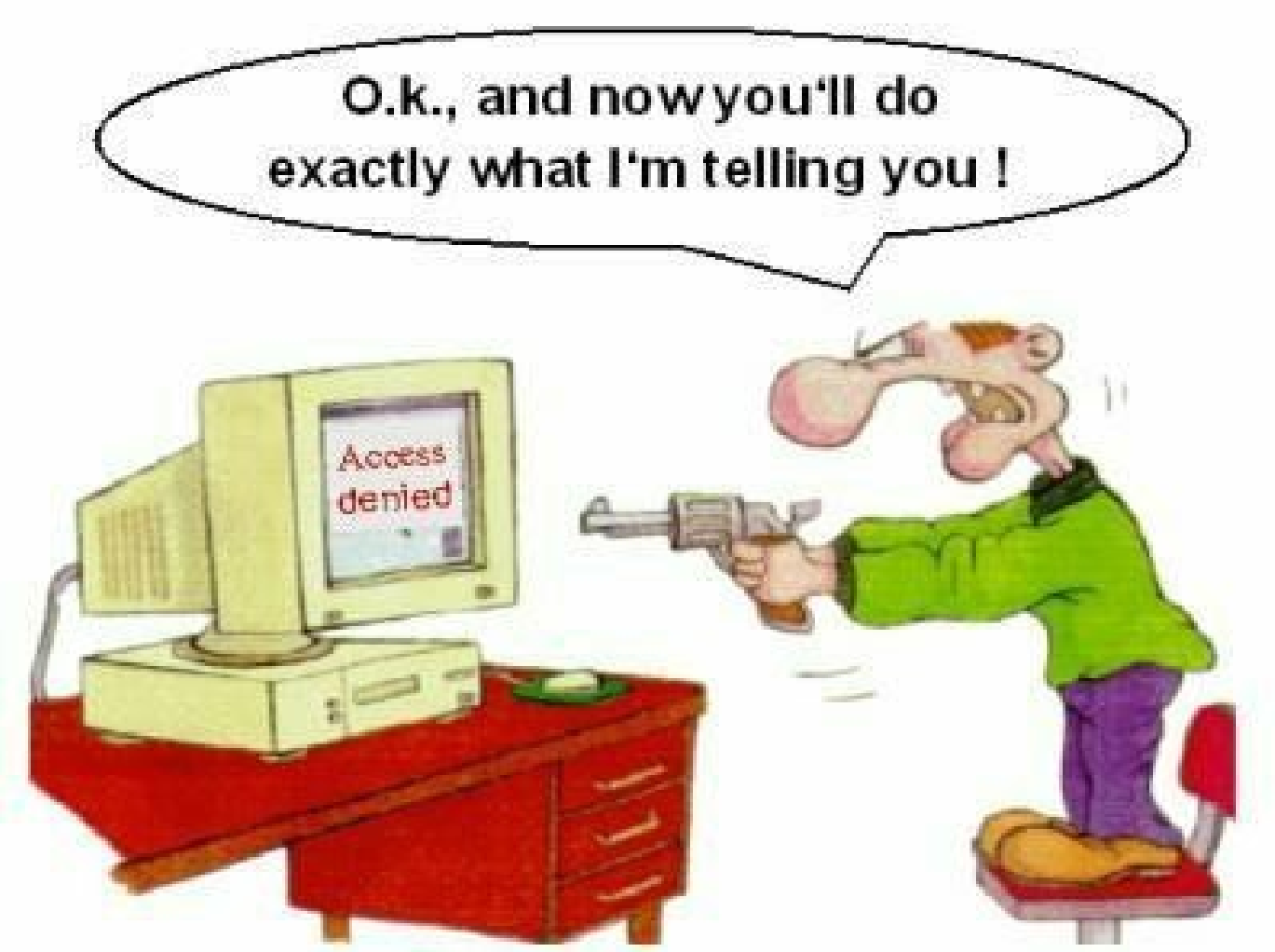
Example 5

Special Internal Macros

- Each of the following four macros is only expanded just before it is used. So the meaning of the macro may vary as the `makefile` progress

$?	List of dependencies changed more recently than the current target
$@	Name of the current target
$<	Name of the current dependency
$*	Name of the current dependency w/o extension

Things To Do



If you have problems visit me in counseling hours. . . .