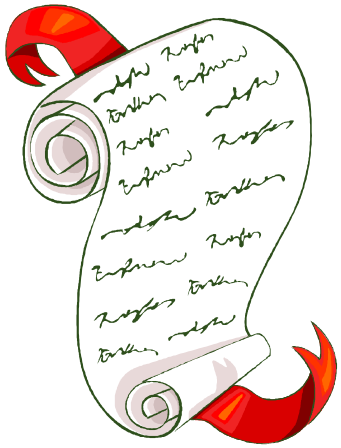
Lecture # 1.2
UNIX make Utility & Packages

Course: Advanced Operating System

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)
- Binary vs Open-Source s/w Packages
- Downloading and installing open-source softwares
- Packaging your software projects using
  - **`GNU autotools`**
  - **`Cmake`** utility

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

Binary Softwares
vs
Open-Source Softwares

Binary Software Packages

- A binary package is a collection of files bundled into a single file containing
  - executable files (compiled for a specific platform),
  - man/info pages,
  - copy right information,
  - configuration and installation scripts
- It is easy to install softwares from binary packages built for your machine and OS, as the dependencies are already resolved
- For the Debian based distributions (Ubuntu, Kali, Mint, ArchLinux) they come in **`.deb`** format and the package managers available are `apt`, `dpkg`, `aptitude`, `synaptic`
- For RedHat based distributions (Fedora, CentOS, OpenSuse) the packages come in **`.rpm`** format and the available package managers are `rpm` and `yum`.

Open-Source Software Packages

An Open-source software is a software with its source code made available with a license in which the copyright holder provides the rights to study, change, and distribute the software to anyone and for any purpose (GNU GPL). Normally distributed as a tarball containing:

- Source code files
- README and INSTALL
- AUTHORS
- Configure script
- Makefile.am and Makefile.in

A source package is eventually converted into a binary package for a platform on which it is configured, build and installed. We normally use source packages to install softwares for following reasons:

- We cannot find a corresponding binary package
- We want to enhance functionalities of a software
- We want to fix a bug in a software

Downloading and Installing Open-Source Softwares

How to download OSS Packages?

- **Option 1:** You can download from some ftp repository using either your browser or may be the famous `wget` command from a Linux terminal.

```
$wget ftp://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz
```

- **Option 2:** You can use advanced packaging tool to download in present working directory by the following command:

```
$sudo apt-get source hello
```

- **Option 3:** You can also use **`git`** if the software is there on some public git repository like **`github.com`** or **`bitbucket.org`**

Magic Spell to Install Open-Source Packages

- A source package is eventually converted into a binary package for a platform on which it is `configured`, `build`, and `install`. Many a times, we all have recited the following magic spell to install a UNIX open-source tarball:

```
$ ./configure
$ make
$ sudo make install
```

Configure the Open-Source Software

```
$ ./configure
```

- The `configure` script makes sure that all of the dependencies for the rest of the `build` and `install` process are available. For example, for a software written in C, it ensures that the system have a C compiler, and find out what it is called and where to find it
- For our **`myexe`** package, once we execute the **`configure`** script it will create the **`Makefile`** that is required to build and install the software
- You can view the contents of the **`Makefile`** of this software, which contains about 3.4K lines

Build the Open-Source Software

$ make

- The build process runs a series of tasks defined in a `Makefile` to build the finished program from its source code. The tarball you download usually doesn't include a finished `Makefile`. Instead it comes with a template called `Makefile.in` and the `configure` script produces a customized `Makefile` specific to your system
- Once you give the **`make`** command, it will run the `Makefile` in the pwd or root directory of package, which may further call other makefiles (if any) in other directories and hence all source files in package will be compiled. This may take some time depending on your system and the size of the software

Install Open-Source Software

```
$ sudo make install
```

•The install process copies the build program and its libraries and documentation to the correct locations. The program's binary(ies) are copied to a directory on your `PATH`, and the program's manual page(s) are copied to a directory on your `MANPATH`. Depending on where the software is being installed, you might need escalated permissions to do this step

Un-installing Open-Source Software

- Makefile has many targets other than **install**, like **uninstall**, **clean** and **distclean**:

```
$sudo make uninstall
$hello
No such file or directory
$which hello
$man hello
No manual entry for hello
```

- The clean target can be used after the installation to remove all the object and executable files from the source directory

```
$ make clean
```

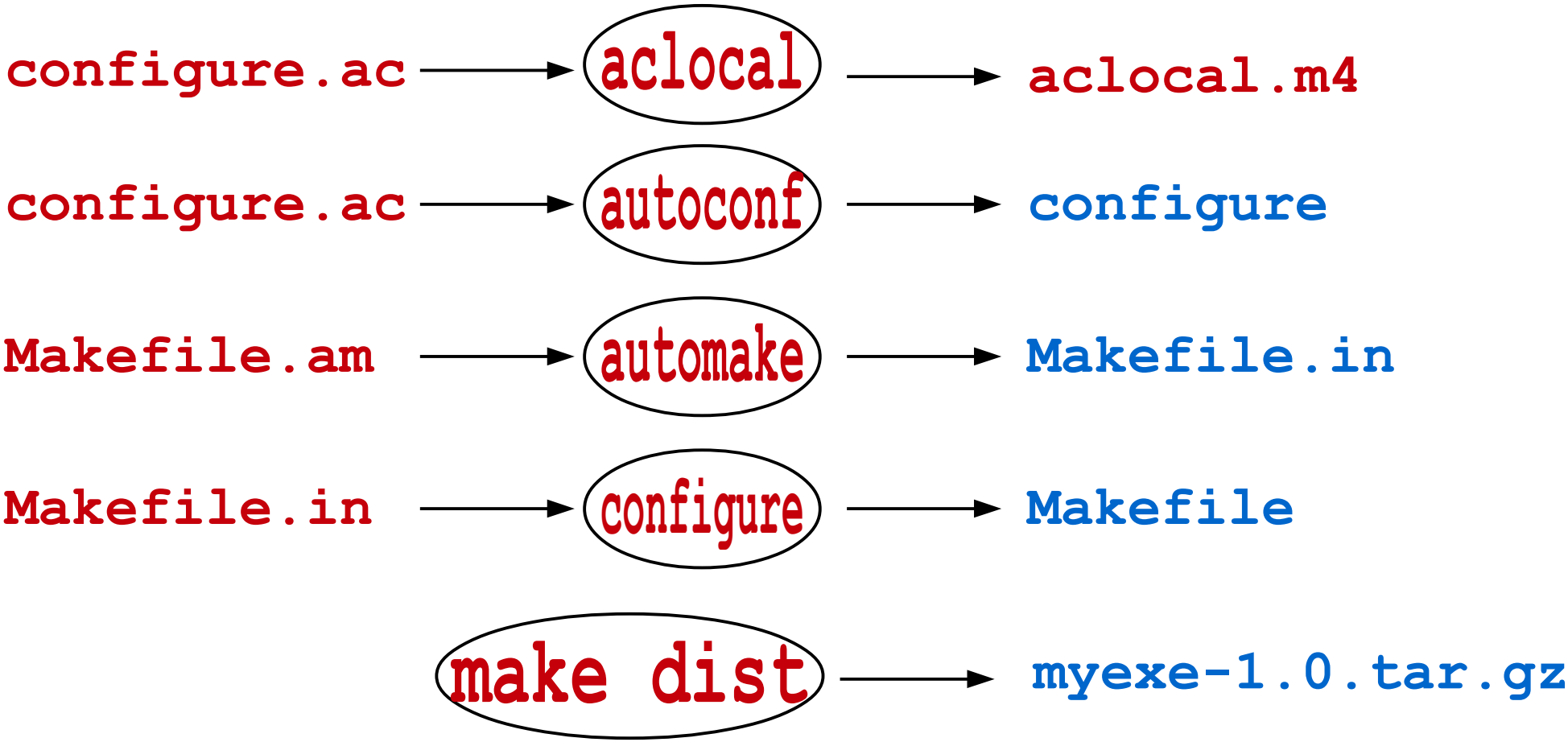
- Similarly, after installation if you want to remove all the files that were created by configure script

```
$ make distclean
```

Packaging your software using GNU Autotools

autoconf & automake

Packaging s/w using GNU autotools



Packaging your software using cmake

What is cmake

In the simplest possible words, `CMake` is a cross platform `Makefile` generator. It is an effort to develop a better way to conf gure, build and deploy complex softwares written in various languages, across many different platforms like Linux, *UNICES, MacOS, MS Windows, iOS, Android,...

https://cmake.org

Friends of cmake

CMake has friends softwares that may be used on their own or together

- CMake: Build system generator
- CPack: Package generator used to create platform-specific installers
- CTest: A test driver tool used to run regression tests
- CDash: A web application for displaying test results and performing continuous integration testing

How Cmake work?

The `CMake` utility reads project description from a file named `CMakeLists.txt` and generates a Build System for a Makefile project, Visual Studio project, Eclipse project, Xcode project, …

Things To Do

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