UNIX Tutorial Seven

7.1 Compiling UNIX software packages

We have many public domain and commercial software packages installed on our systems, which are available to all users. However, students are allowed to download and install small software packages in their own home directory, software usually only useful to them personally.

There are a number of steps needed to install the software.

- Locate and download the source code (which is usually compressed)
- Unpack the source code
- Compile the code
- Install the resulting executable
- Set paths to the installation directory

Of the above steps, probably the most difficult is the compilation stage.

Compiling Source Code

All high-level language code must be converted into a form the computer understands. For example, C language source code is converted into a lower-level language called assembly language. The assembly language code made by the previous stage is then converted into object code which are fragments of code which the computer understands directly. The final stage in compiling a program involves linking the object code to code libraries which contain certain built-in functions. This final stage produces an executable program.

To do all these steps by hand is complicated and beyond the capability of the ordinary user. A number of utilities and tools have been developed for programmers and end-users to simplify these steps.

make and the Makefile

The **`make`** command allows programmers to manage large programs or groups of programs. It aids in developing large programs by keeping track of which portions of the entire program have been changed, compiling only those parts of the program which have changed since the last compile.

The **`make`** program gets its set of compile rules from a text file called **Makefile** which resides in the same directory as the source files. It contains information on how to compile the software, e.g. the optimisation level, whether to include debugging info in the executable. It also contains information on where to install the finished compiled binaries (executables), manual pages, data files, dependent library files, configuration files, etc.

Some packages require you to edit the Makefile by hand to set the final installation directory and any other parameters. However, many packages are now being distributed with the GNU configure utility.

configure

As the number of UNIX variants increased, it became harder to write programs which could run on all variants. Developers frequently did not have access to every system, and the characteristics of some systems changed from version to version. The GNU configure and build system simplifies the building of programs distributed as source code. All programs are built using a simple, standardised, two step process. The program builder need not install any special tools in order to build the program.

The **`configure`** shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a **Makefile** in each directory of the package.

The simplest way to compile a package is:

1. **cd** to the directory containing the package's source code.
2. Type **./configure** to configure the package for your system.
3. Type **make** to compile the package.
4. Optionally, type **make check** to run any self-tests that come with the package.
5. Type **make install** to install the programs and any data files and documentation.
6. Optionally, type **make clean** to remove the program binaries and object files from the source code directory

The configure utility supports a wide variety of options. You can usually use the **--help** option to get a list of interesting options for a particular configure script.

The only generic options you are likely to use are the **--prefix** and **--exec-prefix** options. These options are used to specify the installation directories.

The directory named by the **--prefix** option will hold machine independent files such as documentation, data and configuration files.

The directory named by the **--exec-prefix** option, (which is normally a subdirectory of the **--prefix** directory), will hold machine dependent files such as executables.

7.2 Downloading source code

For this example, we will download a piece of free software that converts between different units of measurements.

First create a download directory

```
% mkdir download
```

Download the software here and save it to your new download directory.

7.3 Extracting the source code

Go into your **download** directory and list the contents.

```
% cd download
% ls -l
```

As you can see, the filename ends in tar.gz. The tar command turns several files and directories into one single tar file. This is then compressed using the gzip command (to create a tar.gz file).

First unzip the file using the gunzip command. This will create a .tar file.

```
% gunzip units-1.74.tar.gz
```

Then extract the contents of the tar file.

```
% tar -xvf units-1.74.tar
```

Again, list the contents of the **download** directory, then go to the **units-1.74** sub-directory.

```
% cd units-1.74
```

7.4 Configuring and creating the Makefile

The first thing to do is carefully read the **README** and **INSTALL** text files (use the less command). These contain important information on how to compile and run the software.

The units package uses the GNU configure system to compile the source code. We will need to specify the installation directory, since the default will be the main system area which you will not have write permissions for. We need to create an install directory in your home directory.

```
% mkdir ~/units174
```

Then run the configure utility setting the installation path to this.

```
% ./configure --prefix=$HOME/units174
```

NOTE: The **$HOME** variable is an example of an environment variable. The value of **$HOME** is the path to your home directory. Just type

```
% echo $HOME
```

to show the contents of this variable. We will learn more about environment variables in a later chapter.

If **`configure`** has run correctly, it will have created a Makefile with all necessary options. You can view the Makefile if you wish (use the **`less`** command), but do not edit the contents of this.

7.5 Building the package

Now you can go ahead and build the package by running the make command.

```
% make
```

After a minute or two (depending on the speed of the computer), the executables will be created. You can check to see everything compiled successfully by typing

```
% make check
```

If everything is okay, you can now install the package.

```
% make install
```

This will install the files into the **~/units174** directory you created earlier.

7.6 Running the software

You are now ready to run the software (assuming everything worked).

```
% cd ~/units174
```

If you list the contents of the units directory, you will see a number of subdirectories.

bin	The binary executables
info	GNU info formatted documentation
man	Man pages
share	Shared data files

To run the program, change to the **bin** directory and type

```
% ./units
```

As an example, convert 6 feet to metres.

```
You have: 6 feet
You want: metres
```

```
* 1.8288
```

If you get the answer 1.8288, congratulations, it worked.

To view what units it can convert between, view the data file in the share directory (the list is quite comprehensive).

To read the full documentation, change into the **info** directory and type

```
% info --file=units.info
```

7.7 Stripping unnecessary code

When a piece of software is being developed, it is useful for the programmer to include debugging information into the resulting executable. This way, if there are problems encountered when running the executable, the programmer can load the executable into a debugging software package and track down any software bugs.

This is useful for the programmer, but unnecessary for the user. We can assume that the package, once finished and available for download has already been tested and debugged. However, when we compiled the software above, debugging information was still compiled into the final executable. Since it is unlikey that we are going to need this debugging information, we can strip it out of the final executable. One of the advantages of this is a much smaller executable, which should run slightly faster.

What we are going to do is look at the before and after size of the binary file. First change into the **bin** directory of the units installation directory.

```
% cd ~/units174/bin
% ls -l
```

As you can see, the file is over 100 kbytes in size. You can get more information on the type of file by using the file command.

```
% file units
```

```
units: ELF 32-bit LSB executable, Intel 80386, version 1, dynamically
linked (uses shared libs), not stripped
```

To strip all the debug and line numbering information out of the binary file, use the strip command

```
% strip units
% ls -l
```

As you can see, the file is now 36 kbytes - a third of its original size. Two thirds of the binary file was debug code!!!

Check the file information again.

```
% file units
```

```
units: ELF 32-bit LSB executable, Intel 80386, version 1, dynamically
linked (uses shared libs), stripped
```

Sometimes you can use the **`make`** command to install pre-stripped copies of all the binary files when you install the package. Instead of typing **`make install`**, simply type **`make install-strip`**

M.Stonebank@surrey.ac.uk, © October 2001