Building programs

LinuxChix-KE

What happens in your CPU?

- It executes a small set of instructions called "machine code"
- Each instruction is just a pattern of bits
- Fast for computer to run
  - Modern processors run at gigahertz clock speeds: billions of instruction cycles per second
- Machine code is hard to read and write!

Let's look at some!

- A machine code program is informally known as "a binary"
- Example: our familiar /bin/ls

  less /bin/ls

  hexdump -C /bin/ls | less
- It's mostly binary code. There may be some text strings which the author included

  strings /bin/ls | less
- We can extract some more info:

  file /bin/ls

  ldd /bin/ls

How are binaries created?

- You *can* write them by hand, using assembly language
  - Done where speed is critical, e.g. some parts of kernel, some games
  - Called "low level" programming
  - Problem: different types of CPU have different machine codes; rewrite same program many times
- Or, you can write in a "high level" language, and then convert this to machine code
- The process is called "compiling"

Points to note

- You have to compile the program before you can run it
- Different systems have their own compilers
  - with care, you can write "portable" software which can be compiled on many different systems
- Compilers are programs themselves
- The machine-code generated by a compiler may not be as efficient as the code written by a good human

High level languages

- 'C' has a close relationship with Unix
  - Unix itself is written in C
  - With some work, Unix can be recompiled ("ported") for different processors
  - Not a very high level: you are still dealing closely with the Unix interface
  - Quite efficient machine code can be generated
- Many other languages exist
  - Higher-level languages can make the programmer's life easier but may run less efficiently
  - Different compiler for each one

Exercise 1: your first 'C' program

- Write a small program in C
- Compile it using the GNU C Compiler (gcc)
- Run it!
- Look at the binary

Source code

- The high-level version you write is called *source code*
- "Open source" (also called “free software”)
  - The author gives you the source code
  - Allows you to compile the software on different platforms
  - Allows you to see how it works and to make modifications
- "Closed source"
  - You are provided with binaries only
  - Very hard to read, very hard to change

Automated building

- A large application has many source files, and many commands are needed to build and link the individual parts
- When you change one source file, it might not be necessary to recompile *everything*
  - would be a waste of time
- So program writers tend to automate building using 'make'

Format of a Makefile

- Contains rules which say:
  - What file you are building (the "target")
  - What sources it comes from
  - The command needed to build it
- These rules are in a file called 'Makefile'
- When you run 'make', it works out which files to rebuild by looking at modification times
  - If any source file has a timestamp later than the target, the target needs to be rebuilt

Exercise 2: your first Makefile

- Create a Makefile for your application
- Change the source, use 'make' to rebuild it
- Run the binary to test

Point to note...

- There are different versions of 'make'
- BSD make and GNU make are different; Linux tends to come with GNU make
- But if you need it, you can install GNU make on your system (package "gmake")

Making portable applications

- There have been many different flavours of Unix released over time
- Programs need subtle changes to compile on different platforms
  - e.g. a particular feature may be available on one system but not another, or may work differently
- Often means changes to the Makefile for different targets
- Programmers can write a different Makefile for each OS... but it's a lot of work!

A solution - auto configuration

- run a shell script before you compile, which tests the system features available and writes out a suitable Makefile
- Most popular example: GNU autoconf
  - program author uses 'autoconf' to write the script
  - the script is called 'configure' and is distributed with the program
  - it takes template files and writes out the final versions (e.g. 'Makefile.in' is rewritten as 'Makefile')

So a typical build operation is:

```
./configure
make
make install    (as root)
```

autoconf also lets you set compile-time options

- e.g. software might normally install in '/usr/local' but you can force it elsewhere

  ./configure --prefix=/opt
- You may also be able to disable or enable certain features in the final application
- Author usually documents the options in an INSTALL or README file

autoconf is not perfect

- It's a shell script
  - not all shells work identically
  - relies on programs in /bin and /sbin (which may vary between OSes)
- It's hard to write your own autoconf script
  - we won't attempt this
  - this is not a programming course anyway
- But generally it works well
  - a lot of work has gone into making autoconf work on a wide range of popular platforms

Where can you find open-source software to install?

- http://freshmeat.net/ is a good index
- Use the web, use mailing lists
- There are advantages of downloading, compiling and building it yourself
  - You can get the very latest versions
  - You have full control over how it is built
- Main disadvantage is it can be difficult to uninstall
  - Might have to locate all the files which were installed, and remove them by hand

The FreeBSD ports collection

- Thousands of programs already catalogued
- Fully automates the process of downloading the source, unpacking, configuring, building, installing
- Applies any necessary patches to work under FreeBSD
- Tends to configure them consistently
  - /etc/rc.conf and /usr/local/etc/rc.d/*
- Records the installed files, so you can remove them using pkg_delete

Disadvantages of ports?

- It's BSD-specific
- Software version in the ports system is not always the latest
- When upgrading one port, you may end up updating others it depends on

FreeBSD packages

- These are actually just ports which have been compiled by someone else
- Easy to install: pkg_add
- No compilation time required
  - Esp. for huge programs: KDE, OpenOffice etc
- However, you don't have any control over compilation options
- Dependency issues
  - package foo-x.x.x requires package bar-y.y.y

FreeBSD itself is open source

- You can install (if you want) the source code
  - /usr/src/sys is the kernel
  - /usr/src/bin is the source for /bin programs
  - /usr/src/usr.bin is the source for /usr/bin programs
  - ... etc
- You can rebuild individual parts, or the whole operating system
  - choose drivers, optimise for your CPU type, ...
- Convenient if only one small component needs to be modified and rebuilt
  - e.g. a security advisory

Rebuilding /bin/ls

```
# cd /usr/src/bin/ls
# make
# make install
```

Rebuilding the kernel

- Can remove unnecessary device drivers to save RAM
  - the GENERIC kernel contains lots of them
  - all drivers are built as loadable modules anyway
- Can add or remove features: e.g. firewalling, IPv6, IPSEC
- Can optimise kernel for your particular CPU
  - GENERIC kernel runs on 486 upwards

Exercise 3

- Configure a new kernel
- Build and install it