Compiling Software

Hands-On UNIX System Administration

2012-02-13
Types of Software Packages

- Programs – things you can run off the command line
- Libraries – software that other source code can use the functions from
- Modules – “extension” code written specifically to work with a certain program
- Script libraries – code archives in languages like Python, Perl, Ruby for various purposes
The Procedure

• Step 0: Download and unpack source
  • Generally, using the `tar` application. e.g.
    • `tar -xvzf MyProg-1.0.tar.gz`
    • `tar -xvjf MyProg-1.0.tar.bz2`

• Step 1: Run `.configure`
  • Prepares source for building on your particular system

• Step 2: Run `make`
  • Compiles source files to binaries (if applicable)

• Step 3: Run `make install`
  • Installs programs and data into system
The Procedure (cont.)

• This works in the majority (70-75%) of cases
• Many other software environments (e.g. scripting languages) have own system
• For example..
  • Python: python setup.py install
  • Perl: perl Makefile.PL; make ...
• When in doubt, look for an INSTALL text file or a README
Patching Software

• When released software has issues, a code patch is released instead of a new version
• Generally come in the unified diff format, which the “patch” utility understands
• You should apply patches before you build, obviously - hence mentioning this here
Example of a Patch

--- maildirtree-0.6/maildirtree.c 2008-10-07 14:19:42 -0700
+++ maildirtree-0.6/maildirtree.c.new 2008-10-07 14:19:48 -0700
@@ -103,7 +103,7 @@
{
    case 'h':
        puts(usage);
-       exits(0);
+       exit(0);
    case 's':
        summary = true;
}
Example of a Patch (cont.)

- Example: `patch -p1 < fix.diff`
- `-p1`: If `fix.diff` wants to look for `a/b/test.c`, actually modify `b/test.c`
- `-p2`: `fix.diff` looks for `a/b/test.c`, actually modifies `./test.c`
- 99% of patches: Enter the source directory, then use `-p1`
Make!

- Powerful build system! You will be using the “GNU” version of make in this class
- Lets you specify what to build, how to build (compiler and arguments), and order to build in
- Includes strong dependency system
- “Don’t build my_program without having libprogram.a built already”
- “If I update foo.h, rebuild foo.c”
Configuring Make

• Configure script generally has options; try ./configure --help
• You can enable features, point it to library install paths that it needs, use different compiler, etc.
• Reacting to a configure/build error often involves trying to find an option that will fix things.
Build Problems

• Missing library:
  • Download, build, and install the needed library

• Missing compiler:
  • Install your OS’s compiler distribution (e.g. Xcode or gcc package on Linux)
  • Make sure to install the C development headers! (e.g. libfoo-dev) on Debian

• Compilation error:
  • Is your operating system supported by the author?
  • You could try and fix it... then submit your solution to the author!
Dependency Hell

• What if your program depends on libfoo?
  • Download libfoo source and try to build
    • libfoo depends on libbar
    • Download libbar source and try to build
    • ... ad infinitum ...

• Many dependencies
• Chains of dependencies
• Conflicting dependencies
• Circular dependencies
• We call this “dependency hell”
How can we (try to) solve this?

- Package systems in Linux distributions (apt, aptitude) or in BSD-type distributions (ports) can help
- Apt-get, aptitude, dpkg
Ports in (Free-)BSD Systems

- Portinstall, portupgrade
  - Installs, upgrades given package
  - Basically runs through configure, make, and make install
  - E.g. portinstall zsh-4.3.15_1
- Portsnap
  - Port snapshot
  - Updates the ports tree
- Pkg_add, pkg_deinstall
  - Adds, uninstalls specified package
  - E.g. pkg_deinstall sudo-1.8.3_1
- pkg_info, pkgdb
  - Gives info about installed packages
  - Manage and search database