

Linux/UNIX utilities for software development

Support for surviving 402

Ted Faber
USC/ISI
20 Oct 2005

What Are We Talking About

programming support

Things to make programming easier:

- automating builds
- facilitating collaboration
- automating tests/analysis
- a little more

Not the work of coding itself:

- compilers
- debuggers
- code organization

Guiding Principles

important safety tips

Computers are good at 2 things:

- remembering things
- performing repetitive tasks

Software development includes plenty of those

- how do I make that program?
- who made that change?
- collect data from 100 runs?

Computers should answer those questions

Larry Wall - first great virtue of a programmer:

Laziness

The Tools of Developers

computers can help!

make

- build software, run test suites, package software

CVS

- ensure consistency, save versions, track development

perl or sh

- automate specific tasks

random stuff

- source code navigation
- C/C++ declaration parsing

Make

track dependencies, do things

Dependency tracker

- when this file is newer than that file, do this

Classic use is keeping programs up to date

- header changes that a source file uses, recompile the source
- object file a library depends on changes relink the library

I use it for many things - this talk has a Makefile

Make Basics

what's in a Makefile

Make recipes:

```
target.o: target.c header.h anotherheader.h
```

```
    cc -c target.c
```

```
target2.o: target2.c header.h
```

```
    cc -c target2.c
```

```
target: target.o target2.o
```

```
    cc -o target target.o target2.o
```

Multiple Dependencies

topological sort

Makefile:

target.o: target.c **header.h** anotherheader.h

cc -c target.c

target2.o: target2.c **header.h**

cc -c target2.c

target: **target.o** **target2.o**

cc -o target target.o target2.o

\$ make target

cc -c target.c

cc -c target2.c

cc -o target target.o target2.o

Making Make Easier

features and utilities

Make features

- variables - parameterize Makefiles
- implicit rules - make knows how
- file inclusion - collect common rules
- fake targets - clean, dist, includes

Utilities

- mkdep (or gcc -M) - automatic dependencies
- imake - create X Makefiles
- autoconf - detect system file locations
- automake - generate GNU Makefiles automatically

A More Featureful Makefile

```
# parameterize the implicit C compilation rule
# .o.c: ${CC} ${CFLAGS} -c -o $> $<
CFLAGS=-O2
# List of all programs to make
ALL=target

# first rule is the default
all: ${ALL}

# implicit rules handle compilation, these just tell dependencies
target: target.o target2.o
target.o: header.h anotherheader.h
target2.o: header.h

# start over rule
clean:
    rm -f *.o ${ALL}

$ make
cc -O2 -c -o target.o target.c
cc -O2 -c -o target2.o target2.c
cc target.o target2.o -o target
```

CVS

versioning system

Tracks changes to a set of files by multiple users

Allows rollback to older versions of files, sets of files

- select by version number, tag, or date
- stores versions as diffs from previous

Conflict detection

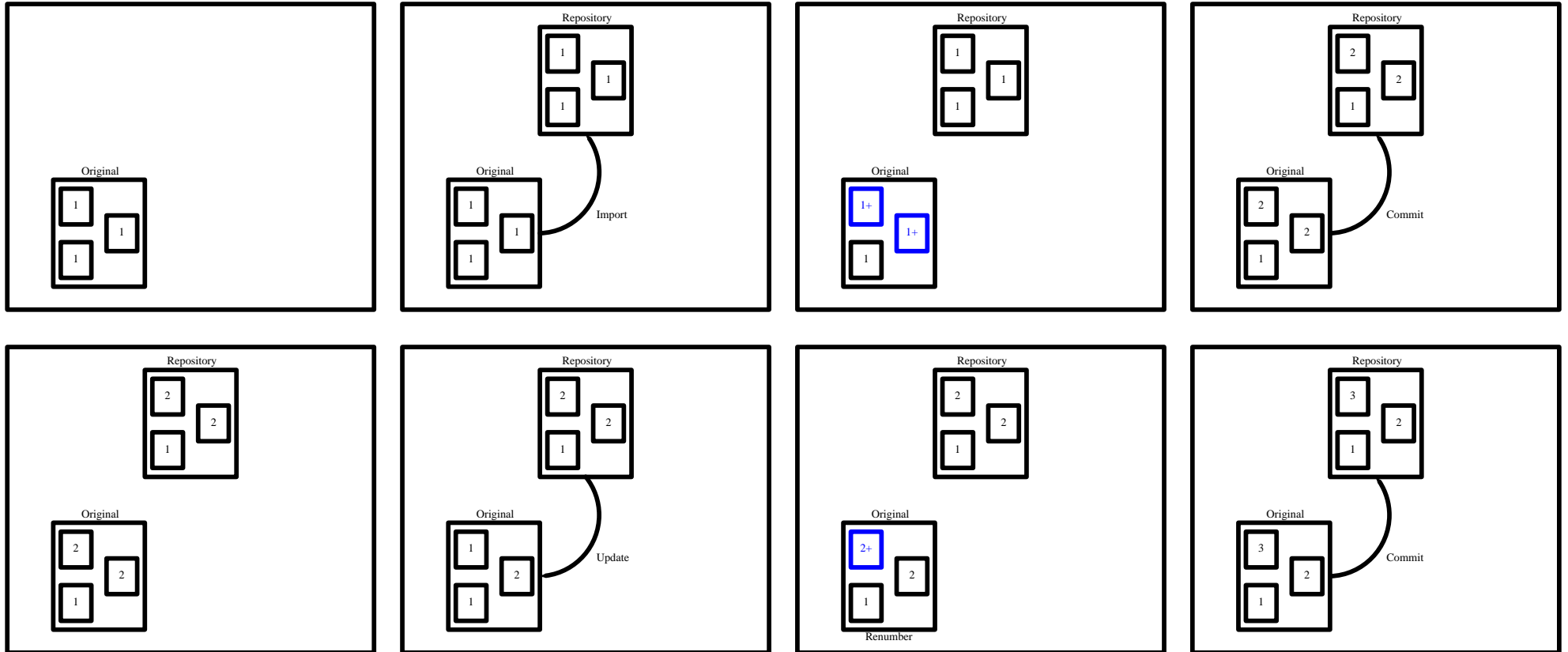
- some automatic resolution

Cheap replication

- integrated with ssh/rsh
- servers for anonymous access

CVS Overview

single user



CVS Overview

multiple users



CVS features

more stuff

Logging

- tells who changed what when

Differences

- tells difference between 2 versions/dates

Replication & Backup

- multiple checkouts for 1 user (CVS_RSH)
- anonymous read-only CVS servers

Automatic Features

- e-mail notifications
- ispell/code indenting

Tasks for Scripting Languages

repetition

Tasks that are ripe for programming, but simple

- do 10 runs of x and collect data
- produce the same format graph from 10 data sets
- run regression tests

Operations too complex to be kept in a Makefile

Modularity

- I may have to average a set of numbers again!

Larry Wall - first great virtue of a programmer:

Laziness

Scripting Languages

seems a shame to fire up the C compiler...

Interpreted languages

- fast prototyping
- easy debugging
- file/filter access

String operations

- translating filenames
- reformatting output

Choices: shells (csh,sh,bash), perl, python

Quick Comparison

scorecard

Shells (bash,csh,sh)

- good file reading/writing support
- good support for connecting external commands
 - pipes, &&, ||
- minimal arithmetic
- minimal string handling

perl, python?

- more verbose file reading/writing
- more control over file operations
- more verbose access to external commands
 - system() calls the shell
 - fork()/exec() syscall interface
- excellent arithmetic and string handling
- more data structures

Ctags

finding things in source

CTAGS creates a jump file to find source code definitions

Most editors (emacs, vi) know how to use this to navigate

Many languages supported:

- Assembly, AWK, App, BETA,
- Bourne/Korn/Zsh Shell, C, C++,
- COBOL, Eiffel, Fortran, Java, Lisp,
- Lua, Make, Pascal, Perl, PHP, Python,
- REXX, Ruby, S-Lang, Scheme, Tcl,
- Vim and Yacc

Cdecl/C++decl

uncross your eyes

Converts English to C/C++ declarations:

\$ cdecl -c declare x as pointer to char

char *x;

\$ cdecl explain char *argv[10]

declare argv as array 10 of pointer to char

\$ cdecl explain int '(*f)[](const char *)'

declare f as pointer to array of function (pointer to const char)
returning int

\$ cdecl explain int '(*f[])(const char *)'

declare f as array of pointer to function (pointer to const char)
returning int

\$ cdecl explain int '(*f)(const char *)[]'

declare f as pointer to function (pointer to const char)
returning array of int

Winding up

tools

make

- <http://www.gnu.org/software/make/manual/>

CVS

- <http://www.nongnu.org/cvs/>

perl or sh

- man sh
- <http://www.gnu.org/software/bash/manual/bash.html>
- man perl
- <http://www.perl.com/pub/q/documentation>

random stuff

- ctags: <http://ctags.sourceforge.net/>
- cdecl: google cdecl rpm