CU CS 4995-03: Internet Systems Programming

Programming Tools

Overview

- source code management: `CVS`
- automation: `make`, automake
- compilation: `gcc`, Visual Studio
- debugging: `gdb`, `truss`
- profiling: `prof`

CVS: Motivation

- problem: lots of people work on same project (code, documentation, specs, . . .)
- stable vs. unstable
- hardware, OS versions
- “feature panic”

CVS: Overview

- version control system (see also RCS or SCCS)
- log of who did what when and why
- collection of directories
- release control
- concurrent revisions: “optimistic”
- network-aware
- single master copy (‘repository”) + local copies

see http://www.cs.columbia.edu/~hgs/cvs

What CVS Isn't/Doesn't...

- build system;
- project management;
- talking to your friends;
- change control:
  - all changes are isolated vs. single logical change;
  - bug fix tracking;
  - track change verification;
- testing program (regression testing) (e.g., `crashme`);
- work flow or process model.

CVS: Setting up a Repository

1. create directory (say) `cvsroot` environment variable or `-d`
2. create `CVSROOT` directory for maintenance files

```
cvsroot
  CVSROOT
    history
    loginfo
    modules
    passwd
  testcvs
    hello.c,v
    Makefile,v
```

`*,v`: history files

CVS: Adding a Module to a Repository

Source files in `src/testcvs`

1. create `cvsroot` directory if necessary
2. `setenv CVSROOT ~/src/cvsroot/`
3. `cd src/testcvs`
4. `import` *rdir vendortag releasetag*: create *rdir* under $CVSROOT. *repository* from current directory, tag *vendortag* for branch, tag *releasetag* for release (generally, “start”); creates branch 1.1.1 with `cvsroot/testcvs/hello.c,v`

```
$ cvs -t import -m "Sample Program"
  testcvs sample start
```

```
N testcvs/hello.c
No conflicts created by this import
```

5. add module name to `cvsroot/CVSROOT/modules`:

```
testcvs   testcvs
something directory/something_else
```

use 'cvs checkout', etc., if you can:

```
$ cvs checkout CVSROOT/modules
$ cd CVSROOT
$ vi modules
$ cvs commit modules
$ cd ..
$ cvs release -d CVSROOT
You have [0] altered files in this repository.
```

```
Are you sure you want to release
(and delete) directory `CVSROOT':  y
```

6. delete working files if you like

CVS: Adding a User

1. `ypcat passwd | fgrep kns10`
2. add user entry to `cvsroot/CVSROOT/passwd`:

   ```
   kns10:ipG1UcZ.djrvI:hgs
   ```

3. add entry to `loginfo` to generate email

   ```
   testcvs  /usr/ucb/Mail -s "%s" kns10 hgs
   ```

4. add entry to `users`

   ```
   kns10:kns10@columbia.edu
   ```

CVS: Using a Repository

As one of the developers, login if on remote server:

```
cvs -d \
  :pserver:kns10@tune.cs.columbia.edu:\
/home/hgs/src/cvsroot login
```

CVS: Using a Repository

Check out the source code files from repository:

```
$ cvs checkout testcvs
cvs checkout: Updating testcvs
U testcvs/hello.c

ls -R
CVS/  hello.c
CVS:
Entries  Repository  Root
```

creates 'working directory' `testcvs` and `CVS` directory

CVS: Committing Changes

1. create or edit file
2. *add* file if new:

```
$ cvs add Makefile
cvs add: scheduling file `Makefile' for addition
cvs add: use 'cvs commit' to add this file permanentl
```

CVS: Committing Changes

commit changes (all files based on modification date):

```
$ cvs commit
RCS file: cvsroot/testcvs/Makefile,v
done
Checking in Makefile;
cvsroot/testcvs/Makefile,v  <- Makefile
initial revision: 1.1
done
Checking in hello.c;
cvsroot/testcvs/hello.c,v  <- hello.c
new revision: 1.2; previous revision: 1.1
done
```

Asks for change message; “sign” it!

CVS: Catching Up

No notification beyond email!

- update local repository

```
$ cvs update
cvs update: Updating .
M hello.c
```

- “merges” changes, may produce “conflicts”.
- output:

U *file*	updated: file not in wd or new file in rep.
M *file*	modified, merged
C *file*	conflict detected, marked by >>>...<<<
? *file*	stray file in wd

CVS: Cleaning Up

```
$ cd ..
$ cvs release -d testcvs
```

Commits any pending changes and then deletes files (if `-d`).

CVS: Viewing Differences

Differences between version checked out and working copy:

```
$ cvs diff hello.c
Index: hello.c
===================================================================
RCS file: /home/hgs/src/cvsroot//testcvs/hello.c,v
retrieving revision 1.2
diff -r1.2 hello.c
2a3,4
> /* The simplest program in C. */
>
7c9
<   exit(0);
---
>   exit(1);
```

CVS: Revisions

- each revision increases rightmost number by one: 1.1, 1.2, …
- more than one period ⇒ branches
- versions of file = CVS *revisions*
- (released) versions of software = CVS *releases*
- new file gets highest first digit
- `cvs commit -r 2.0`: makes all revisions to 2.0

CVS: Revision Tagging

Use `cvs tag` to tag revisions (software release)

```
$ cvs tag rel-0 hello.c
T hello.c
$ cvs status -v hello.c
File: hello.c   Status: Locally Modified
Working revision:    1.2  Wed Sep  8 17:50:20 1999
Repository revision: 1.2  /cvsroot/testcvs/hello.c,v
Existing Tags:
 rel-0        (revision: 1.2)
 start        (revision: 1.1.1.1)
 sample       (branch: 1.1.1)
```

CVS Branches

- released (stable) vs. development (unstable) version
- *branch* on revision tree for released version
- `cvs tag -b rel-1-fix`
- `cvs rtag -b -r rel-1 rel-1-fixes testcvs`

CVS History

```
$ cvs annotate hello.c
Annotations for hello.c
***************
1.1 (hgs 08-Sep-99): #include <stdio.h>
1.1 (hgs 08-Sep-99):
1.1 (hgs 08-Sep-99): int main(int argc, char *argv[])
1.1 (hgs 08-Sep-99): {
1.1 (hgs 08-Sep-99):   printf("hello world!\n");
1.2 (hgs 08-Sep-99):
1.2 (hgs 08-Sep-99):   exit(0);
1.1 (hgs 08-Sep-99): }
```

Use `loginfo` to set notifications.

CVS Notifications

- `cvs status` reports status
- watch certain files for modifications:

```
cvs watch on hello.c
cvs watch off hello.c
```

- creates read-only files
- need to run `cvs edit` before editing

Which File is This?

- find out in binary the version used
- `$Log$`

```
static char *id="@(#) $Id$"
```

becomes on checkout

```
static char *id="@(#) $Id: hello.c,v 1.3 1999/09/08
  18:58:50 hgs Exp $"
```

- `ident a.out` or `what a.out`:

```
$Id: hello.c,v 1.4 1999/09/08 19:00:59 hgs Exp $
```

gcc

- `gcc -o hello hello.c` also links
- `gcc -c hello.c` produces `hello.o`
- `-Wall`
- `--help`
- `-PIC`: position-independent code
- `-E`: pre-process only
- `-L`: load libraries (e.g., -lm loads `/usr/lib/libm.so`)

ldd and nm: What's linked?

Dynamic libraries:

```
$ ldd a.out
libc.so.1 =>      /usr/lib/libc.so.1
libdl.so.1 =>     /usr/lib/libdl.so.1
```

Content of archive or executable:

```
$ nm [-g] a.out
0004155b R charmap
00030f10 t cleanfree
0002e304 W close
0002d9dc T close_fd
```

Creating a Library

- *static* library for linking: `libsomething.a`
  - create .o files: `gcc -c helper.c`
  - `ar rlv libsomething.a *.o`
  - `ranlib libsomething.a`
  - use library as `gcc -lsomething`, searches in `/usr/lib`, etc.
- *dynamic* library
  - `gcc -shared -fPIC helper.c -o libhelper.so`
  - use same as above, `LD_LIBRARY_PATH`

make

- maintain dependency graphs: source ← object, LaTeX ← PostScript, data ← graph, ...
- based on modification times (problems?)

```
target ... : dependency
    command
    command
    ...
```

Check editor (tabs vs. spaces)!

make

```
all: hello clean

clean:
        rm -f *.o

helper.o: helper.c

OBJ = helper.o \
  hello.o

hello: $(OBJ)
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $(OBJ)
```

make variables

`$@`	name of current target
`$?`	list of dependencies newer than target
`$<`	name of dependency file
`$*`	base name of current target
`$%`	for libraries, the name of member

implicit rules, e.g., a .c file into .o:

```
.c.o:
$(CC) $(CFLAGS) $<
```

make depend

```
depend: $(CFILES) $(HFILES)
	$(CC) $(CFLAGS) -M $(CFILES) > .state
# Disgusting hack to include the dependency state file
# in both GNU make and BSD 4.4 make.  Explanation:  BSD
# make uses CPP-style # directives; GNU make considers
# anything starting with a # to be a comment.  So get
# each to ignore the directive for the other.
#if 0
include .state
#endif
#include ".state"
```

autoconf

- automatically check for system dependencies
- `autoconf` $\longrightarrow$ `configure`
- uses macro processor `m4`
- configuration script `configure`
- `#define` .h files
- uses small compiled programs

autoconf structure

`AC_INIT(file)` ; check for file
checks for programs
checks for libraries
checks for header files
checks for typedefs
checks for structures
checks for compiler characteristics
checks for library functions
checks for system services
`AC_OUTPUT(Makefiles)` ; generate Makefiles

`autoscan`: examine sources, generate `configure.in`

autoconf library checking

```
AC_CHECK_LIB (library, function,
  [if-found, [if-not-found]])
AC_CHECK_LIB(socket, socket,
        AC_DEFINE(HAVE_LIBSOCKET)
        LIBS="-lsocket $LIBS"
        SOCKLIBS="-lsocket $SOCKLIBS")
```

automake

- generate Makefiles automatically, based on autoconf
- `Makefile.am` $\longrightarrow$ `Makefile.in`
- “flat”: all files in single directory
- “deep”: top-level only configuration
- “shallow”: flat, with libraries in subdirectories

automake

1. prepare `Makefile.am`
2. prepare `configure.in`, adding
   `AM_INIT_AUTOMAKE(rtptools, 1.13)`

```
$ aclocal; automake -a; autoconf
adds missing files; to compile:
\begin{verbatim}
$ ./configure; make
```

autoconf example

```
dnl Process this file with autoconf to
dnl produce a configure script.
AC_INIT(multimer.c)
AM_INIT_AUTOMAKE(rtptools, 1.13)

dnl Checks for programs.
AC_PROG_CC
AC_PROG_CPP
AC_PROG_INSTALL
```

Autoconf Example, cont.

```
dnl Checks for libraries.
AC_CHECK_LIB(socket, nsl, yp_match)

dnl Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS(fcntl.h malloc.h sys/time.h unistd.h)
```

Autoconf Example, cont.

```
dnl Checks for typedefs, structures,
dnl and compiler characteristics.
AM_C_PROTOTYPES
AC_C_CONST
AC_C_BIGENDIAN
AC_TYPE_UID_T
AC_TYPE_OFF_T
AC_TYPE_PID_T
AC_TYPE_SIZE_T
AC_HEADER_TIME
AC_STRUCT_TM
```

Autoconf Example, cont.

```
dnl Checks for library functions.
AC_PROG_GCC_TRADITIONAL
AC_TYPE_SIGNAL
AC_FUNC_STRFTIME
AC_FUNC_VPRINTF
AC_CHECK_FUNCS(gethostname gettimeofday mktime \
  select socket strdup strerror \
  strstr strtol uname)
```

Autoconf Example, cont.

```
dnl Substitute for AC_CHECK_LIB().  First check for
dnl function, then check for function in library
AC_DEFUN(JAPHAR_CHECK_LIB,
[AC_CHECK_FUNCS($2)
if test "$ac_cv_func_$2 = no"; then
  AC_CHECK_LIB($1, $2, [ac_cv_func_$1=yes
     LIBS="-l$1 $LIBS"]
  )
fi])

JAPHAR_CHECK_LIB(socket, socket)
```

Autoconf Example

```
# Until code uses WORDS_BIGENDIAN
if test x"$ac_cv_c_bigendian" = xyes; then
        AC_DEFINE(RTP_BIG_ENDIAN)
else
        AC_DEFINE(RTP_LITTLE_ENDIAN)
fi

AC_OUTPUT(Makefile)
```

defines `-DRTP_BIG_ENDIAN=1`

Makefile.am

```
## Process this file with automake
## to produce Makefile.in
AUTOMAKE_OPTIONS = foreign
bin_PROGRAMS = rtpdump rtpplay rtpsend rtptrans
COMMON = \
 ansi.h host2ip.c hpt.c multimer.c \
 multimer.h notify.c notify.h \
 rtp.h sysdep.h types.h vat.h
rtpdump_SOURCES = $(COMMON) rd.c rtpdump.h rtpdump.c
rtpplay_SOURCES = $(COMMON) rd.c rtpplay.c
rtpsend_SOURCES = $(COMMON) rtpsend.c
rtptrans_SOURCES= $(COMMON) rtptrans.c
EXTRA_DIST = ChangeLog
```

gdb

- must compile with -g to get symbol table
- `gdb a.out` or `gdb a.out core`
- `gdb a.out 1234` attaches to process 1234
- source commands:

list main.cc:12	show source
p(rint) x	show variable x
where	where am I in call stack?

- does not work (well) across threads

gdb: execution

run arg	run program
call $f(a, b)$	call function in program
step N	step N times *into* functions
next N	setep N time *over* functions
up N	select stack frame that called current one
down N	select stack frame called by current one

gdb: break points

How do they work?

break main.cc:12	set break point
break foo	set break at function
clear main.cc:12	delete breakpoint
info break	show breakpoints
delete 1	delete break point 1
display x	display variable at each stop

Debugging Front-Ends

- Data Display Debugger (DDD): GUI for gdb, dbx, jdb, …

Other Tools

code generation	`lex, flex`	lexical analysers
	`yacc, bison`	compiler generator
memory	`purify`	memory leak detector
	`ElectricFence`	leaks, out-of-bounds
performance	`prof`	profile performance
	`truss`	system calls
bug tracking	`gnats`	
	`bugzilla`	

Others: regression testing, user input simulator, bug tracking, …

Dynamic memory allocation

- Unix:

  p = malloc(n): allocate memory,

  p = calloc(n, s): allocate and clear aligned memory

  p = realloc(p, n): increase/decrease allocation; copy content

  ```
  int x[];

  x = calloc(10,sizeof(int));
  x[5] = 7;
  x = realloc(x, 12 * sizeof(int));
  x[11] = 8;
  ```

  free(p): release memory

alloca(n): allocate memory on stack (can't free, realloc)

- typically, list of buffers
- heap and stack divided by *break address*
- sbrk call ➠ change data segment
- getrlimit(): get memory limit

ElectricFence

- overruns `malloc()` boundary
- touch (read, write) memory release by `free()`
- place inaccessible (VM) memory page after each allocation
- uses `mmap()` with /dev/zero and `mprotect()`
- "Mapping a zero special file creates a zero-initialized unnamed memory object of a length equal to the length of the mapping and rounded up to the nearest page size as returned by sysconf."

ElectricFence

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char *s = malloc(5);

  printf("uninitialized: %s\n", s);
  strcpy(s, "A very long string");
}
```

Testing:

```
$ gcc -g test.c libefence.a -o test
```

```
$ ./test
Electric Fence 2.0.5
uninitialized:
Segmentation fault (core dumped)
$ gdb test core
(gdb) where
```

runs fine without ElectricFence!

`prof`: profiling

- profiling = execution profile of call graph
- periodic CPU sampling

```
$ gcc -pg nested.c -o nested
$ nested
$ gprof nested gmon.out
  %   cum  self              self    total
 time seconds s    calls  ns/call ns/call name
38.27 1.24 1.24     1000 1240.00 2040.00  outer
35.49 2.39 1.15                             internal_mcount
24.69 3.19 0.80 10000000    0.08    0.08  inner
 1.23 3.23 0.04     1000   40.00   40.00  _write
 0.31 3.24 0.01     1000   10.00   50.00  _doprnt
 0.00 3.24 0.00     2000    0.00    0.00  _realbufend
```

`truss`

(Linux: `strace`)

- show execution trace of *system* calls
- does not show stdio calls
- `-p`: attach to existing process id
- `-f`: follow children
- `-u libc:` also follow user libraries

```
$ truss -u libc -d test
```

also: `top` to show memory utilization

purify

check for

- memory leaks
- overruns
- access to free'd memory
- open file descriptors

```
$ purify -cache-dir=/tmp/purify gcc -g test.c
$ a.out
```