

CS810 - Advanced Programming in the UNIX Environment

—

UNIX development tools

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann

`jschauma@cs.stevens.edu`

`http://www.cs.stevens.edu/~jschauma/810-APUE/`

Software Development Tools

UNIX Userland is an IDE – essential tools that follow the paradigm of “Do one thing, and do it right” can be combined.

The most important tools are:

- the compiler toolchain, obviously
- `make(1)` – project build management, maintain program dependencies
- `diff(1)` and `patch(1)` – report and apply differences between files
- `cvs(1)` – distributed project management, version control
- `gdb(1)` – debugging your code

cc(1) and ld(1)

The compiler usually performs preprocessing (via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

Different flags can be passed to `cc(1)` to be passed through to each tool as well as to affect all tools.

The order of the command line flags *may* play a role! Directories searched for libraries via `-L` and the resolving of undefined symbols via `-l` are examples of position sensitive flags.

The behavior of the compiler toolchain may be influenced by environment variables (eg `TMPDIR`, `SGI_ABI`) and/or the compilers default configuration file (MIPSPro's `/etc/compiler.defaults` or `gcc's specs`).

make(1)

make(1) is a command generator. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
- performs *selective* rebuilds following a *dependency graph*

make(1)

make(1) is a command generator. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
- performs *selective* rebuilds following a *dependency graph*
- allows simplification of rules through use of *macros* and *suffixes*, some of which are internally defined

make(1)

make(1) is a command generator. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
- performs *selective* rebuilds following a *dependency graph*
- allows simplification of rules through use of *macros* and *suffixes*, some of which are internally defined
- *macros* can be overwritten on the command-line

make(1)

`make(1)` is a command generator. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
- performs *selective* rebuilds following a *dependency graph*
- allows simplification of rules through use of *macros* and *suffixes*, some of which are internally defined
- *macros* can be overwritten on the command-line
- different versions of `make(1)` (BSD `make`, GNU `make`, Sys V `make`, ...) may differ (among other things) in
 - variable assignment and expansion/substitution
 - including other files
 - flow control (for-loops, conditionals etc.)
 - special targets

Priority of Macro Assignments for `make(1)`

1. Internal (default) definitions of `make(1)`
2. Current shell environment variables. This includes macros that you enter on the *make* command line itself.
3. Macro definitions in *Makefile*.
4. Macros entered on the `make(1)` command line, if they follow the *make* command itself.

diff(1) and patch(1)

diff(1):

- compares files line by line
- output may be used to automatically edit a file
- can produce human “readable” output as well as diff entire directory structures
- output called a *patch*

patch(1):

- applies a diff(1) file (aka *patch*) to an original
- may back up original file
- may guess correct format
- ignores leading or trailing “garbage”
- allows for reversing the patch
- may even correct context line numbers

cv_s(1)

The Concurrent Versions Systems, a version control system similar to *RCS* or *SCCS*, allows you to

- keep old versions of files
- keep a log of who, when, and why changes occurred
- handle multiple developers, multiple directories
- perform release engineering by creating *branches*

cv_s(1) does have some shortcomings:

- impossible to rename a file (must remove and create new file → loss of revision history)
- commits are not atomic

Links

CVS:

- <http://www.netbsd.org/images/graphs/release-branches.gif>
- <http://www.cvshome.org/docs/manual/cvs-1.11.6/cvs.html>
- <http://subversion.tigris.org/>

Other:

- <http://www.netbsd.org/Documentation/elf.html>