Laboratory of Operating Systems and Software Design

Introduction to Linux
First steps of C programming

Ing. Roberto Farina

roberto.farina@cefriel.it

Summary

- Introduction to Linux and tools overview
- First steps in C
  - Write and compile the first program
  - Automation tools: `make`
  - Debug tools: `gdb`
- Online documentation
- Argument list
- Standard I/O
- Assert and error handling
- Environment variables
- Temporary files
- Appendixes

Linux introduction

- **Linux** is "just" the kernel
- GNU provided the standard C library, the compiler and other tools needed to develop applications

- ...GNU/Linux!

- What's needed for system programming:
  - A configured and running Linux kernel
  - The standard C library
  - A compiler
  - An editor to type the code

Gcc

- **http://gcc.gnu.org**

- **THE** compiler on Linux systems (and others...)

- Translates code written in an high level language (*human readable*) in object code (*machine understandable*)

- Includes different tools
  - ▶ **cpp0**, **cc**, **as**, **ld**

- Supports many programming languages
  - ▶ C, C++, Java, ...

First example

- **`gcc -c main.c`**
  - The result is **main.o** (object code)
- **`g++ -c reciprocal.cpp`**
  - The result is **reciprocal.o** (object code)

- What does **`-c`** means?

- Linking:
- **`g++ -o reciprocal main.o reciprocal.o`**

Useful options

- **-I**
  - Directories where to look for header files
- **-D**
  - Defines macros or symbols (**#define** in the code)
- **-Ox**
  - Optimization level: useful when compiling the release version
- **-l**
  - Libraries to use for linking
- **-L**
  - Directories where to look for libraries

make

- Very simple idea:
  - ▶ I want to indicate what to compile and how to do it
  - ▶ I want to indicate dependencies too
- Solution: **Makefile**

- Targets must be defined to identify what you want to build
- Every target has a specific action associated with it

- Special target: **clean**
  - ▶ Removes generated files

Debug: gdb

- It is necessary to recompile files adding debug information
  - `make CFLAGS=-g`
- `gcc` includes additional information in object files and executables
- `gdb` use these information to associate operations to the specific line of code

- Main actions: **run**, **where**, **up**, **break**, **print**

Documentation

- **Man pages**
  - Different sections
    - 1: user commands
    - 2: system calls
    - 3: standard library functions
    - 8: system and amministration commands
- **Info**
  - More detailed than man
- **Header files**
  - **/usr/include**
- **Man pages report header files and libraries to use for each function**

Argument list

- A way to interact with the execution environment
  - Enables parameter passing from outside
- **argc** and **argv** as parameters of *main* function
  - *argc* is an int which represents the number of elements in the argument list
  - *argv* is an array of char*
    - Its dimension is argc
    - Array elements refer to the argument list elements
    - The first element is the program name

© 2006 - R. Farina

Command line options

- Two types
  - Short options
    - Single dash and one char
    - Quick to use
  - Long options
    - Two dashes and a name
    - Easier to read (script) and to remember
- Usually programs accept both types of options
  - Same options in both forms (-h, --help)

getopt_long

- **Short options**
  - A string lists valid options
- **Long options**
  - A specific data structure with details about valid options
- **Scanning loop**
  - `getopt_long` returns next option in the list
    - -1 when at the end of the list
  - Take the specific action for the option
  - Option arguments must be read too!

Standard I/O

- Standard input and output streams used by **scanf**, **printf** and other library functions
  - **stdin** and **stdout**
  - Redirection and pipelining to concatenate programs
- Standard error: **stderr**
  - Where error messages and warnings should be printed

  fprintf(strerr, "Error: …")
- These streams ca be used with low level functions too
  - read, write, …

Stream properties

- **stdout** is buffered
  - ► Data are written to the console when the buffer is full
  - ► To force the flush
    `fflush(stdout)`

- **stderr** isn't buffered
  - ► Data are directly written to the console

assert

- Macro to verify conditions at runtime

    `assert(pointer != NULL)`

    - If it's not verified, it produces an error message

        `Assertion 'pointer != ((void *)0)'  failed.`

- The argument must be a boolean expression

- Useful as source code documentation

- Asserts represent a cost in terms of performance

    - Useful to debug but to avoid in the release version

- Solution:

    - Option `–DNDEBUG` to gcc when compiling

        - `assert` are discarded during pre-processing

assert

- How to use

```
for (i = 0; i < 100; ++i)
{
  int status = do_something();
  assert(status == 0);
}
```

- How **NOT** to use

```
for (i = 0; i < 100; ++i)
    assert(do_something() == 0);
```

Error handling

- Almost each system function returns 0 when successful and a non zero value in case of error
  - There aren't standards about the return value in case of errors
    - Always read man pages!
- **errno**: a special variable storing information in case of errors
  - When an error occurs, the system sets this variable to a value that describes the error type
  - Each system call uses it
    - Immediately copy the value in a backup variable!

errno

- Error codes are integer values
- Possible values defined by macros
  - ▶ Each one starts with an "E"
  - ▶ **<errno.h>** must be included
- **strerror** returns a string describing the error
  - ▶ **<string.h>** must be included
- **perror** prints the error message directly on **stderr**
  - ▶ **<stdio.h>** must be included

Environment variables

- **`environ`**: special global variable defined by GNU C Library
  - ▶ char**: array of string pointers
  - ▶ Strings are in the form *NAME=value*
  - ▶ Modification of **`environ`** is not suggested!

  `extern char** environ;`

- An alternative: **`getenv`**
  - ▶ Returns the value of the specified variable
    - • **`NULL`** if the variable is not defined

  `getenv("VARIABLE_NAME");`

Temp files

- Useful to store big amounts of data or to communicate with other programs

  - E.g.: to free resources

- Possible problems:

  - More than one instance of the running program -> unique names to avoid data loss

  - Critical data -> correct access rights must be set

  - Possible attacks if file names are known -> not predictable

mkstemp

- Creates a temp file with a name derived from a template
  - File name ends with XXXXXX
- Returns a descriptor to use with the write functions family
- File are not deleted automatically
- If the file is intended for internal use `unlink` must be called
  - Remove the directory entry corresponding to the file
  - Files are reference-counted
    - Not removed until it is closed

tmpfile

- Used when the file is not intended for communication with other programs

- Returns a file pointer

- `unlink` automatically invoked
  - The file is removed after calling `fclose`

Bibliografy

- Advanced Linux Programming
  - www.advancedlinuxprogramming.com
- Kernighan, Ritchie – The C language
- Man pages
- Info

Laboratory of Operating Systems and Software Design

Appendix 1: a make alternative

Ing. Roberto Farina

roberto.farina@cefriel.it

A different building strategy

- Usual steps to compile and install a program
  - configure, make, make install
- Autotools failed in one of their main goal: portability!
- Scons: a Software CONStruction tool
  - Written in Python
  - Simplified approach
  - Portable across platforms

Laboratory of Operating Systems and Software Design

Appendix 2 : editors

Ing. Roberto Farina

roberto.farina@cefriel.it

Emacs vs. VI

- **Emacs is (probably) the most famous editor in Linux**
  - Complex but powerful
- **Non standard on UNIX systems**
  - Solaris or (some version of) BSD doesn't provide it

- **vi**
  - Every UNIX-like OS provides it
  - A plugin architecture allows to extend the set of functions it provides

Emacs

- **C-x C-f** to open a file
- **C-x C-s** to save a file
- **C-x C-c** to quit Emacs
- **Tab** to correctly format the code
- Syntax highlighting
- Development tools integration
  - It is possible to use compilers and debuggers directly from the editor
- Customizable using LISP
- With graphical version it is not needed to remember key shortcuts

vi

- `vi [+line] file...`
  - Load files
    - `+line`   Places the cursor to line `line`
- `vi` works on
  - Chars
  - Words: sequenze di chars delimitate da spazi
  - Lines: sequenza di chars delimitata da `<CR>`
  - Blocks: consecutive lines
- `vi` has two main modes
  - `ex` comands
  - `vi` comands

vi: ex commands

:[<s>,<e>]<cmd>[<mod>] [<arg>]

- **<cmd>** — Main command (1 char)
- **<mod>** — Command modifier (1 char)
- **<arg>** — Arguments and options
- **<s>,<e>** — Line set to work on

- Commands must be terminated with **<CR>**

- Last line of the terminal shows:
  - The file name, the line number and the number of chars
  - **ex** commands while they are entered

`:q`   Quit

   `:q!`   Forced quit

`:w`   Write

   `:w!`   Forced save

   `:wq`   Save and quit

   `:w <f>`   Save the file with the new name `f`

`:x`   Write and quit

   `:x <f>`   Save the file with the new name `f`

`:e`   Edit

   `:e <f>`   Load the file `f`

:r   Read: loads a file in the current one

    :r <f>   Insert the file **f** in the current position

:<l> Goto: moves to the specified line

    :<l>   Goes to line # **l**

    :$ Goes to the last line of the file

`vi`: `vi` commands

[*<n>*]*<cmd><mod>* [*<text>*]

- *<cmd>*   Main command
- *<mod>*   Command modifier
- *<n>*   Repeat the command n times
- *<text>*   Text to be inserted

- Must be closed with an *<ESC>*
- Classified in:
  - Positioning commands
  - Insert commands
  - Delete commands
  - Modify commands
  - Search commands

`vi`: positioning commands

←↑→↓   Move the cursor

      <n>h   **n** chars left

      <n>l   **n** chars right

      <n>j   Up **n** lines

      <n>k   Down **n** lines

^   Move to the beginning of the current line

$ Move to the end of the current line

G   Move to the end of the current file

vi: insert commands

i,I   Insert

        i*<t>*   Insert *t* in the current position

        I*<t>*   Insert *t* at the beginning of the line

a,A   Append

        a*<t>*   Insert *t* after the current position

        A*<t>*   Insert *t* at the end of the line

o,O   Open

        o*<t>*   Insert *t* in the previous line

        O*<t>*   Insert *t* in the next line

x,X   Delete character

        <*n*>x   Delete *n* chars to the right

        <*n*>X   Delete *n* chars to the left

vi: delete commands

d   Delete entity

      dd   Current line

      <n>dd   *n* lines starting from the current one

      d$, D   From current pos. to the end of the line

      d^   From current pos. to the beginning of line

      dG   From current pos. to the end of the file

      dH   From current line to the beginning of the page

      dL   From current line to the end of the page

      <n>dw   *n* words after

      <n>db   *n* words before

vi: modify commands

y   Yank entity

    <*n*>yy   Copy *n* lines

    <*n*>yw   Copy next *n* words

    <*n*>yb   Copy previous *n* words

    y$ Copy until the end of the line

    y^   Copy from the beginning of the line

    yG   Copy until the end of the file

vi: modify commands

p   Paste

r,R   Replace

      r<c>   Replace the current char

      R<t>   Overwrite the text

~   Case

      ~   Change the case of the current char

      <n>~   Change the case of next **n** chars

.   Redo

      <n>.   Repeat **n** times the last command

u   Undo

vi: search commands

/   Find forward

      /<ere>   Search for the regexp *ere*

      //   Search next occurrence

?   Find backward

      ?<ere>   Search for the regexp *ere*

      ??   Search next occurrence

n,N   Next occurrence

      n   Search in the same direction

      N   Search in the opposite direction