System Administration

Alessandro Barenghi

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

alessandro.barenghi - at - polimi.it

March 17, 2015

Introduction

Why a system administration lesson?

- Strong binding between system architecture and network stack
- System administration and management skills are required to “survive” in this environment
- As a bonus, they come in handy in a lot of other contexts
- They are taken for granted in other courses

Chosen Platform

Why Linux?

- The chosen platform for the course is GNU/Linux
- No restrictions on the redistribution of tools/practice material
- The notions easily generalise to affine Unices (f.i. MacOS X) with minor changes
- Any recent Linux distribution is fine for practicing

Study methodology

The four letter creed

- "Ten minutes of direct practice are worth ten hours of study in system adminstration"
- Pick a distribution and install it in a realistic environment (real Iron is the best choice)
  - Debian is an easy shot for beginners
  - Slackware is extremely clean as far as internal structure goes
  - Gentoo might not be for the faint of heart, but it's really effective as far as learning goes
- Begin practicing soon, these notions take time to consolidate
- Linux is endowed with an outstanding manual suite available typing `man <command>` from a terminal emulator

Overview

What you should already know

- How to perform basic operations from the commandline interface (list files, change directory, copy files)
- Basic knowledge of the OS from Computer Architecture and OS course (what is a process, OS inner workings)
- Basic knowledge of the underlying hardware, from the same course (how does a context change take place)
- Solid knowledge of the C language fundamentals: the whole Linux kernel and commandline utilities are written in C

Overview

Lesson contents

- How to manage the multitasking environment in Linux
- How to examine what a program is employing as resources
- How to inspect a process running on the system
- How to manage a running system in times of trouble

Commandline interface

The shell

- We will be using a commandline interface to perform all our tasks as it is the simplest interface
- The commandline interpreter, a.k.a. the shell is a program which runs an infinite loop where:
  1. The commands typed in are read and tokenized (= divided in strings, splitting on spaces) when we press the return key
  2. The first token is the name of the program which should be executed, the others are its parameters
  3. The shell performs a `fork`, and its child `exec`s the program with the proper parameters
  4. The shell `wait`s for the end of the execution of the child, and then accepts a new command

Under the hood

Process Tree Structure

- In a Linux system the processes are bound by a strict parent-son family relationship
- The boot process, after the kernel has bootstrapped the machine, yields the control to either `init` or `systemd`
- The `init` or `systemd` process generates all the other system process either directly (via `fork`, and `exec`s) or indirectly
- Every running process, except `init` or `systemd` has a father: it's the process which he was forked from
- Every process has a unique numeric identifier called Process ID (PID): on Linux it's represented as an 16 bit integer

Seeing processes

What is currently running?

- A typical task is to inspect a system to examine which processes are running
- This can be done through the `ps` command
- `ps` provides a list of the running processes, together with related information (e.g. process status, PID)
- A visual representation of the family tree of all processes can be obtained with `pstree`

Common `ps` options

Proper use of `ps`

- `ps` supports multiple syntaxes for the options, we use the standard one
- `-e` shows every process running
- `-u <user>` shows all the processes running as a certain user
- `-Lf` shows the number of threads of every process
- `a` shows the processes belonging to any user
- `x` allows to see processes which are not bound to a terminal

Interactive listing

A live view of the system

- `ps` provides a static snapshot of the running processes
- In a number of situations it is more helpful to see the evolution of the system state
- The `top` command provides a sequence of dynamic snapshots
- `htop` is an enhanced version of top with more information
- Both tools periodically refresh the list of processes on screen
    - Basically, they keep obtaining the same information as `ps`

How do they work?

A(n old) system introspection filesystem

- The information read by `ps`/`top`/`htop` comes from the `proc` filesystem
- It is a virtual filesystem: nothing is present on the disk
- When a program tries to list the contents of something in the `proc` filesystem, the OS generates these contents from scratch
- Provides a file-based interface to OS-level informations
- It's Linux specific, but other Unices provide equivalent mechanisms to access the same pieces of information

Managing running processes

Running in the background

- Running a command from the shell results in the shell waiting for its completion: this is known as running in foreground
- `CTRL-C` aborts the foreground execution instantly
- `CTRL-Z` stops the foreground execution, preserving its state
- Typing `bg` with a stopped program runs it in the background
- Typing `fg` with a program running in the background, brings back the execution to the foreground
- Adding an `&` at the end of a command starts the execution in the background

Process Inspection

Analyzing a live process

- We now know how to inspect which processes are running
- Up to now, the processes were (almost) black boxes
- Time to open the box and see what's inside
- This can be done via:
  - Debuggers (`gdb`)
  - Process tracers (`strace`, `ltrace`, `lttng`)
  - File monitoring tools (`lsof`)

Inspecting the execution of a program

The GNU Debugger

- The GNU Debugger provides a plethora of functions to inspect the inner working of a program
- It acts through running the process under exam and tracing its behaviour via the `ptrace` system call
- It is able to alter the memory content of the program at the human debugger's will
- You should already be familiar with its working from the first programming course

Monitoring syscalls

Coarser grain in monitoring

- An alternative to per-instruction debugging is analysing the process at system call level
- Every process[a] needs to interact with the operating system
- It is possible to monitor the parameters and return values of every system/library call performed by a process
- Two tracing tools are available `strace` (for system calls) and `ltrace` (for library calls)

[a] or at least any process doing meaningful tasks

Monitoring syscalls

`strace`

- Follows the execution of a process and monitors syscalls, attaching to it via a `ptrace` call
- `strace` by default prints out *all* the syscalls of a process
- Since they usually are a *LOT* `-o <filename>` redirects to a file :)
- `-e=group` allows you to select only some syscalls relative to a peculiar function
  - `process`: syscalls concerning process management (e.g. fork)
  - `network`: syscalls concerning network (e.g. connect)
  - `file`: file read/write syscalls, fseek
  - `signal`: signal firing and masking calls

Reducing the clutter

Useful options

- The `-p <PID>` options allows you to attach to a running process[a]
- The `-f` option enables the tracing of the child processes alongside the father
- The `-t` option prints out the system time at which the syscall has been run

[a] provided you have the permission to do so

Monitoring dynamic library calls

`ltrace`

- Follows the execution of a process and monitors dynamic library calls
- `ltrace` by default prints out *all* the library calls of a process
- Shares most options with `strace`, so you can remember them easily
- Only traces calls to dynamically linked libraries, no way to distinguish the ones to statically linked ones

An overlook on files

A common interface

- Under UNIX everything is abstracted as a file
- The prime interface for data communication between kernelspace and userspace, and among processes are files
- This implies that all the physical devices are seen as a file by the programs in userspace
- Moreover, also sockets are seen as a peculiar type of file
- Although the library calls are compatible, it is strongly advised not to mix them (e.g. use `write` instead of `send`) on a socket

An overlook on files

Monitoring open files

- A well designed file monitoring tool is a prime resource to understand what's happening
- The ultimate tool for file (i.e. mmapped devices, libraries, sockets and so on) monitoring is `lsof`
- The basic use just lists *all* the open files on a system
- Depending on the compile time options, `lsof` may list only the files of the processes owned by the user

Reducing clutter, once again

`lsof` options

- the `-c <string>` option allows to list all the files opened by any command starting with `<string>`
- the `-c /<regex>/` option allows to list all the files opened by any command starting with `<regex>`
- the `+D` option allows to list all open files in a directory
- the `-u` option allows to list all open files of a certain user
- the options are usually combined with a logical *OR*
  - `-a` switches to *AND* combining

Not only files

Monitoring special files

Remember, "Under unix everything is a file":

- So we can also easily list open and listening sockets!
- the `-i @IP` option allows to list all the sockets open from-to a certain IP address
- the `-P` option prints numeric ports representations
- the `-p` option allows to list all open files from a precise PID
- the options may be reversed through prepending the usual `^`

Managing the running processes

Interacting with the system

- Up to now we have seen how to investigate the behaviour of a running system
- We did not alter it, we just observed what was going on
- This was done at system level (process tree examination) and at a finer grain (single process examination)
- We will now see how to manage the running processes

Asynchronous communication

Signals

- The prime mechanism in a Unix system to communicate asynchronous information to a process are signals
- Signals can be though of as “software generated interrupts”
- Every process has a signal handlers table acting as the interrupt handler table
- The signal handler may choose to ignore the signal, do something or just fall back to the default action
- Usually the default action is the termination of the process

Signals

Here's a list of commonly used signals, together with the default behaviour:

- `SIGTERM` : terminates the process "gracefully" (file buffers are flushed and synchronized)
- `SIGSEGV` : terminates the process, issued upon a segfault
- `SIGQUIT` : terminates the process dumping the memory segment into a `core` file
- `SIGKILL` : terminates abruptly the execution [unstoppable]
- `SIGSTOP` : sets the process in wait state [unstoppable]
- `SIGCONT` : resumes the execution of a process

Issuing signals by hand

The `kill`

- The commandline tool to send signals is aptly named … `kill`
- Common syntax: `kill <signal> [options]`
- The signal to be sent can be specified either by its ID or its textual mnemonic
- The issued signals set flags in the fired signal table of the target process
- Since signals are resolved when a process is going to be run, `STOP` then shoot signals to die-hard processes
- Resume them with a `SIGCONT` and they'll be gone