Introduction to System Administration

Grado en Informática. 2025/2026

Departamento de Computación
Facultad de Informática
Universidad de Coruña

Antonio Yáñez Izquierdo

Contents I

Contents II

The role of the System Administrator

The role of the System Administrator
→system administrator

who is the system administrator?

- For a system to function properly it is necessary that the users have their privileges restricted
  - They should not be able to affect other users' files and processes
  - They should not be able to tamper with the system configuration or affect the system performance in any way
- Certain tasks still need to be done: it is necessay that one (or some) user(s) have the required privileges to carry them out
- This user (or set of users) is (are) commonly refered to as the *system administrator*

what's up with the system administrator account?

- there are two different approaches to implementing the concept of system administrator
  a. the system has a specific account for performing system administration tasks
  b. some user accounts have the privileges required to perform administrative tasks
- the *a)* approach is the one typically found in UNIX systems: the administrator account is the **root** account
- the *b)* approach is the one typically found in *windows* systems: there are *limited* accounts and *administrator* accounts. There is also an account labeled `administrator` which is, obviously an *administrator* account

rights and roles

- there is yet another approach: certain users can perform certain administrative tasks
  - example: a user can modify the network configuration but cannot add/remove software packages
- this can be accomplished
  - making specific groups own the files and programs neccesary to perform such administrative tasks and adding those users to these groups
  - through the more sophisticated *rights and roles* paradigm as used in the *solaris system* and its derivatives (*opensolaris . . .*)
  - through the command *sudo* and the *sudoers* file
- In these systems there is still an account, the **root** account, with the privileges to perform all the administrative tasks and with ability to give or deny other users administrative rights

The role of the System Administrator
→tasks of the system administrator

things to do

- the following tasks usually fall under system administrator responsability
  - **installation**: install/upgrade the system and the software applications
  - **configuration**: configure the system and the software applications so that
    - the system fuctions as efficiently as possible
    - the users can use the system in the right way
  - **maintenance**: the system and the software applications need to be functioning properly
  - **security**: system must be kept safe (both from unintentional mistakes and from malicious attacks)

doing what has to be done

- carrying out the aforementioned jobs usally requires performing some (or all) of the following tasks
  - adding/deleting user accounts
  - installing/upgrading/removing hardware
  - installing/upgrading/removing applications
  - doing backups and restoring files from backups
  - tuning system parameters
  - monitoring system activity
  - keeping up with security notices and patches
  - writing/rewriting/finding *scripts* to automate as many tasks as possible
  - talking to users and other system administrators

how to do what has to be done I

- Although there are some fancy-looking programs to perform administrative tasks,
  - yast on SUSE linux
  - `system-config-*` on fedora linux
  - SMIT (System Management Interface Tool) on AIX
  - `sysinstall` on freeBSD
  - Solaris Management Console on Solaris
  - Gnome System Menus
  - ...

how to do what has to be done

- much of the system administration is done modifying text files and running text mode programs
  - many servers don't even have a graphical screen connected to them
- the S.A. should know what commands can be used and which files need to be modified
- it is important to understand how things work, otherwise unexpected things can happen
  - for example: never run a script for configuring the firewall from a ssh session

how to do what has to be done

- planning must be done before actually doing anything
- if possible make changes one at a time
- if possible, make changes reversible
  - keeping copy of the original configuration files
  - commenting lines out instead of deleting
  - commenting lines include in configuration files
  - ...
- test before running (specially important in *scripts* running as *root*)

Users and groups

Users and groups
→users

what is a user?

- user accounts are the mean by which *real world users* present themselves to the system and are granted (or denied) access to it
- *authentification* is the process by which the system verifies that a user is who he/she claims to be
- a user in the system is an entity that can own files and execute programs (thus creating processes). It **may or may not** be a *real person*

privilege separation

- some users (sometimes known as *pseudousers*) exist only to execute specific services and own the files associated with those services.
  - **Example:** users *www-data* and *sshd* run the web and ssh servers respectively, but are not associated with any individual person
  - This is done to increase system security: if the services were to be run by the *root* user and had some security issue that could be exploited, the root account-and thus the whole system, would be compromised. This way only the www-data (or sshd) account would be compromised in case such situation arised
- This is often refered to as *privilege separation*

username and UID

- each user has a name that identifies it, called *username*
- when adding a user, the system administrator has to provide both a username and a user identification number (UID)
- the system uses the UID (not the username) internally. The username is just *mapped* to the UID.
- when adding a user the system administrator also assigns this user to one or more groups

Users and groups
→groups

what is a group?

- a group is a collection of users gathered together for *whatever reason*
- a group is identified by a groupname and internally by a **G**roup **ID**entification number, GID
- one group can have one or more users. Users are said to *belong* to that group
- one user can belong to one or more groups, although one of them is called the *primary group* of the user: the one defined in the `/etc/passwd` (or equivalent) file
- the user and group behind the execution of a process determine which files in the system the process can access

Users and groups
→user and group definition files

user and group definition files

- the user and group information of the users **defined locally** in one system resides in the following ASCII text files

`/etc/passwd` this file defines the user acounts in the system. One line per user, constituted by fields separated by :
On older systems the *encrypted password* (strictly speaking, the result of crypting a base text using the password as key) was stored here as well. Example:

```
root:x:0:0:the_almighty_system_administrator:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
```

`/etc/shadow` (only on newer systems) pasword definition file, one line per user. Example

```
root:$6$pz0jXkuY$6M71ZfZk1ecQv...sxodXCCZh5CVeR.DxQ1bOHn37t5OL.:14578:0:99999:7:::
daemon:*:14578:0:99999:7:::
```

user and group definition files

`/etc/group` group definition file, one line per group. Example

```
wheel:*:0:root,antonio
daemon:*:1:daemon
```

- Some systems have specific files that are not found on other systems

  /etc/gshadow only on linux
  /etc/master.passwd only on BSD systems

Files, processes and devices

Files, processes and devices
→files and directories

files and directories

- everything stored in the system is a file
  - a C program text
  - some letter
  - the executable file that constitutes a comand
  - the list of users in the system (`/etc/passwd`)
  - ...
- security on unix depends greatly on access to files
- files are organized in a hierarchical structure. Looks like a *tree* but it is actually a graph
- this structure has a root directory designed by /

file permissions and ownership

- Each file in the system is owned by both ONE user and ONE group
  - the user owning the file may belong to several groups, but the file is owned only by one group
- The file has three sets of permissions associated (usually called the mode of the file)
- each set of permissions is a subset of the word **rwx**
  - the letter indicates the permission is granted
  - the - sign instead the letter indicates the permission is not granted

file permissions and ownership

- The first set are the permissions for the user owning the file, the second set the permissions for the group owning the file and the third set the permissions for the rest of the users in the system
  - r the file can be read: view the file contents
  - w the file can be written: modify the file contents, that is, the file can be appended, modified, overwritten ...
  - x the file can be executed

file ownership and permissions

- example

```
-rw-r----- 1 antonio audio 4656065 Sep 13 13:06 audiofile.mp3
```

- this file is owned by user `antonio` and group `audio`, its permissions are `rw-r-----` (the first `-` indicates it's a regular file)
  - the first set of permissions, `rw-`, means that a process from user antonio can read and write to the file
  - the second set of permissions, `r--`, means that a process from any user belonging to group audio can read the file
  - the third set of permissions, `---`, means that the rest of the users in the system can't read the file, nor write to it, neither execute it (were it an executable file)

permission representations

- the permissions are represented by an octal three digit number
  - one octal digit per set of permissions
  - to obtain the binary value for each of permissions we use `1` to represent the permission is granted and `0` otherwise
- that way `rw-r-----` is represented as binary `110 100 000` and octal 640
- `rwxr-xr--` would be represented as binary `111 101 100` and octal 754

special permissions

- there are three more special permissions
  - **sticky bit** (octal 1000) (`--------t`). On modern systems it has no effect on files, on older systems, if one executable had the sticky bit set, the system would not deassign the swap space after executing the file
  - **setgid** (octal 2000) (`-----s---`) The process excuting a file with the setgid bit set gets the group credential of the group owning the executable
  - **setuid** (octal 4000) (`--s------`) The process excuting a file with the setuid bit set gets the user credential of the user owning the executable

example of special permissions

- Consider the file

  ```
  -rwsr-sr-x 1 antonio audio 4656065 Sep 13 13:06 program1.out
  ```

- its permissions are rwsr-sr-x (binary 110 111 101 101, octal 6755)
- processes from user antonio can read write and execute the file
- processes from users belonging to group audio can read and execute the file
- processes from any user can read and execute the file
- a process executing the file gets its user credential changed to that of user *antonio* and its group credential changed to that of group *audio*

permissions in directories

- the permissions in directories have the following meaning

r	The directory can be listed: see the names of the files in it
w	The contents of the directory can be modified: files can be added to it or files can be removed from it
x	The files in the directory can be accessed
setgid	Files created in this directory get owned by directory group instead of the group of the process creating the file
sticky bit	Only the owner (or one who has write acess) of a file can delete it, even having write access to the directory

Files, processes and devices
→other types of files

other types of files

- In addition to files and directories unix supports the following types of files
  - **block devices**. They have no assigned space, just two numbers (major a minor) used to tell the kernel which device driver to use when accessing the device
  - **character devices**
  - **symbolic links** A file whose *contents* are the path to the file the link refers to
  - **fifo** A first in first out file

other types of files

- non symbolic links are not special files, they are just another name to an existing file
- the comand `ls -l` lets us distinguish the different types of files in a unix system

```
abyecto:/home/antonio/pru# ls -l
total 12
brw-r--r-- 1 root root 15,  3 Sep 13 18:02 block_device
crw-r--r-- 1 root root  9, 51 Sep 13 18:14 char_device
-rw-r--r-- 2 root root     93 Sep 13 18:03 file
-rw-r--r-- 2 root root     93 Sep 13 18:03 link
lrwxrwxrwx 1 root root      4 Sep 13 18:18 symlink -> file
drwxr-xr-x 2 root root   4096 Sep 13 18:01 this_is_a_directory
prw-r--r-- 1 root root      0 Sep 13 18:03 this_is_a_fifo
```

Files, processes and devices
→commands for dealing with files

usual commands to access files

- these are the most usual commands to access files in a unix system. The online *man* page is the ultimate source of information in the system we're using

`mv` moves (or renames) a file or directory

`cp` copies files or directories

`chown` changes the owner of a file (must be root)

`chmod` changes the mode (permissions) of a file (must be owner)

`chgrp` changes the group of a file (must be owner)

`mkdir` creates a directory

usual commands to access files

`mknod`	creates a special file (directory, device or fifo)
`mkfifo`	creates a special fifo file
`ln`	creates a link (both symbolic and non symbolic)
`rm`	removes a file
`rmdir`	removes a directory
`ls`	lists the contents of a directory
`cd`	changes directory
`umask`	sets the file creation mask (default permissions)

Files, processes and devices
→processes

processes

- a process is the entity the Operating System has to execute a program
- in UNIX a process is identified by a number, its PID (Process IDentification)
- except for process with pid 1 (`init` or `systemd`) and some special processes created at boot time, a process is always created by another process, its *parent process*, in what we usually call *forking*

processes

- thus, unix systems have a tree like process structure where process 1, is the common ancestor of all processes
- when a process terminates it returns a value (with information on how it has terminated) that can be retrieved only by its *parent process*
  - if in a *script* (or a shell) this can be obtained with the variable '**$?**'

process credentials

- the system uses what we call *process credentials* to determine which user and group are responsible of the execution of a process
- each credential pair consists of a user credential and a group credential, that we call the uid and gid of the process
- there are three pairs of credentials: real, effective and saved, so one process has real uid, real gid, effective uid, effective gid saved uid and saved gid.
- the effective credentials are used to determine the preivileges (which files can be accessed . . .); the real credentials represent the real user behind the process (thet are used to decide from which processes signals can be received); saved credentials indicate which changes ob the efective credentials con be made.

types of processes

- **interactive processes**: they are run interactively from a terminal, this terminal is called the controlling terminal or controlling *tty* of the process
- **non interactive processes**: often called *batch processes*, for example processes submited to execution via the *at* o *cron* commands. They lack controlling terminal
- **daemons**: system processes usually initiated at boot time, that run continously providing different services. For example the *log process*. Some unixes have the utility `start-stop-daemon` to initiate them

Files, processes and devices
→processes and programs: the path

processes and programs

- a program consists of one or more files on disk that contain executable code
- in order to execute a program a process must exist. Either
  - a new process is created
  - an existing process replaces its code and executes a program
- new processes can be created from a process, so one program can actually create more than one process

processes and programs

- when we want to create a new process to execute a program we can
  - from a graphic environment: click (or double click) in the appropiate place
  - from the shell's command line: type the name of the file we want to execute
- if we want to execute a program from the shell's command line it is not enough to type the name of the file we want to execute: **the complete pathname of the file must be typed**
  - if we want to execute the program `xterm` which resides in the `/usr/bin` directory we should type

    ```
    $ /usr/bin/xterm
    ```

the path

- executable files can be placed anywhere in the filesystem
  - however, the executable files supplied with the OS are placed only in certain directories (`/bin`, `/usr/bin`...)
- to avoid unnecessary typing, a list of directories where executables can be found is provided via the environment variable PATH,
  - if the `/usr/bin` directory is included in the PATH enviroment variable, to execute the program `xterm` we can just type

    ```
    $ xterm
    ```

the path

- the PATH is an environment variable
  - each process inherits it from its parent process
  - to add (or remove) directories from it we must use the command `export` in *sh*, *ksh*, *bash* ...
  - we can modify it with `setenv` in *csh*-like shells
- for security reasons the current working directory, "." should not be included in the PATH, or at least be included as the last directory

Files, processes and devices
→signals

signals

- signals are a way of notifying a process of certain events: contrl-C, commnunication terminated, ...
- signals are sent to a process when the event occurs. Upon receiving a signal a process can
  - terminate
  - do nothing (signal is ignored)
  - perform a specified action (signal is catched). For example, many unix daemons re-read their configuration file when they receive the HUP signal (signal HUP is catched)

signals

- a signal can also be *blocked* (will not be attended until it is *unblocked*)
- signals can also be sent directly to a process via the command `kill`. Thus we have a method to terminate, stop or continue processes
- The most used signals are

HUP	hangup
STOP	stop the process (can not be catched, blocked ot ignored)
KILL	terminate process (can not be catched, blocked ot ignored)
CONT	continue process (can not be catched, blocked ot ignored)

Files, processes and devices
→commands to dealing with processes

basic commands to dealing with processes

`top`	display system processes
`ps`	get info on the system processes
`kill`	send a signal to a process (usually causing its termination)
`fg`	bring to the foreground a process in the background
`bg`	continue a process in the background
`ptree`	(`pstree` on some systems), lists the process tree

basic commands to dealing with processes

`batch`	submit for non interactive execution
`at`	submit for execution at a specified time
`nice`	execute a program at a different priority. Some systems have also specific utilities depending on their priorities policies, such as `priocntl` (solaris), `chrt` (linux) or `rtprio` (freeBSD)
`renice`	change a running program priority
`ionice`	change a running program disk i/o priority (linux only)

Files, processes and devices
→devices

devices

- unix treats devices like files.
- each device has a file in the filesystem, typically in the */dev* directory
- devices can be block devices (for example disks) or character devices (for example terminals)
- the device file has no allocated space, instead two numbers:
  - major number: which device driver in the kernel handles the device
  - minor number: how to access the device.

example of device numbers

- The following listing shows that in linux the different partitions of different hard disks use the same major number (the same device driver handles disks)

```
abyecto:/dev# ls -l sd*
brw-rw---T 1 root disk 8,  0 Sep 17 09:34 sda
brw-rw---T 1 root disk 8,  1 Sep 17 09:34 sda1
brw-rw---T 1 root disk 8, 10 Sep 17 09:34 sda10
brw-rw---T 1 root disk 8, 11 Sep 17 09:34 sda11
brw-rw---T 1 root disk 8, 12 Sep 17 09:34 sda12
brw-rw---T 1 root disk 8,  2 Sep 17 09:34 sda2
brw-rw---T 1 root disk 8,  3 Sep 17 09:34 sda3
brw-rw---T 1 root disk 8,  4 Sep 17 09:34 sda4
brw-rw---T 1 root disk 8,  5 Sep 17 09:34 sda5
brw-rw---T 1 root disk 8,  6 Sep 17 09:34 sda6
brw-rw---T 1 root disk 8,  7 Sep 17 09:34 sda7
brw-rw---T 1 root disk 8,  8 Sep 17 09:34 sda8
brw-rw---T 1 root disk 8,  9 Sep 17 09:34 sda9
abyecto:/dev#
```

other example of device numbers

```
crw------- 1 root root    4,  0 Sep 17 09:34 /dev/tty0
crw-rw---- 1 root tty     4,  1 Sep 17 09:35 /dev/tty1
crw------- 1 root root    4, 10 Sep 17 09:34 /dev/tty10
crw------- 1 root root    4, 11 Sep 17 09:34 /dev/tty11
crw------- 1 root root    4, 12 Sep 17 09:34 /dev/tty12
crw------- 1 root root    4, 13 Sep 17 09:34 /dev/tty13
crw------- 1 root root    4, 14 Sep 17 09:34 /dev/tty14
crw------- 1 root root    4, 15 Sep 17 09:34 /dev/tty15
....
crw-rw---T 1 root dialout 4, 64 Sep 17 09:34 /dev/ttyS0
crw-rw---T 1 root dialout 4, 65 Sep 17 09:34 /dev/ttyS1
crw-rw---T 1 root dialout 4, 66 Sep 17 09:34 /dev/ttyS2
crw-rw---T 1 root dialout 4, 67 Sep 17 09:34 /dev/ttyS3
```

devices

- device files can be created with the command *mknod*. *mknod* receives the major a minor numbers as arguments
- a script called MAKEDEV exists in the dev directory to help create the device files
  - this way the command '`./MAKEDEV audio`' from within the */dev* directory would create the audio devices with the appropriate major and minor numbers
- many modern unix systems have a dynamic device management service that creates the device files at boot time (*devfs*, *udev* . . .) whereas traditionally they were created at installation time. In these systems there is no '`MAKEDEV`' script

Becoming superuser

why become superuser

- when we have to perform administrative tasks we must become superuser to gain the necessary privileges to do so
- we must work as superuser **only** the time necessary to perform the administrative tasks
  - if an application only runs properly for the superuser and it is not an administrative application then the application is not correctly installed, probably due to some file permission issues
- there are three ways to become superuser
  - login as root
  - use the `su` command
  - use the `sudo` command

Becoming superuser
→loging in as root

login as root

- we can login as root directly at the console or via ssh
  - login as root (or as any user for that matter) using *telnet*, *rlogin* ... is strongly discouraged as the communication is not encrypted and the root password would travel in the clear making it possible to someone in the network to gain access to it
- as an aditional security precaution we can disable login as root from the console (or any terminal for that matter), or via ssh, this way to become root one has to
  - know the root password
  - have a valid account on the system

disable login as root

- the files `/etc/securetty` (linux), `/etc/ttys` (BSD) or `/etc/default/login` (solaris) enable or disable root login from certain terminals
- in newer versions of *Solaris* `root` is a *role* so it cannot login directly
- stating *PermitRootLogin no* in sshd configuration file disables root login via ssh

Becoming superuser
→the su command

using the su command

- the '**su**' command allows one user to substitute his/her identity (provided he/she knows the appropriate password)
- '**su**' without any arguments substitutes the invoking user's identity by that of the `root`
- '**su** -' provides the root environment as well
- '**su**' would prompt for the root password and, should the authentication be correct, fork a shell with superuser privileges
- as an aditional security measure on BSD systems, a user must belong to the *wheel group* to successfully become root via the `su` command

Becoming superuser
→the sudo command

using the sudo command

- the '**sudo**' command allows a permitted user to execute a command as the superuser or another user
- the file '`/etc/sudoers`' implements the security policy
- some systems disable the *root* account and are administered via the '**sudo**' command. Example: *ubuntu*
  - any user belonging to the '**adm**' group can perform any task as root via the '**sudo**' command providing just his/her own password
  - the user created during system installation is made a member of the '**adm**' group
  - to enable the *root* account in these systems, setting the *root* password is enough

Basic system administration commands

basic system administration commands

- in addition to the commands shown on previous sections for dealing with files and processes, there are some commands the system administrator should be familiar with. The following ones allow managing the user and group databases

`useradd`	adds a user to the system
`adduser`	adds a user to the system
`userdel`	removes a user from the system. Some systems have also `rmuser`
`usermod`	modify a user definition on the system
`groupadd`	adds a group to the system. Some systems also have `addgroup`
`groupmod`	modify a group definition on the system
`pw`	manipulate user and group databases (freeBSD)

basic system administration commands: vi editor

- As most of the configuration files on the systems are text files, the system administrator should at least be able to do basic file editing with **the one editor present in every unix system: the vi editor**
- this editor has two modes of operation: insertion mode and command mode
  - in insertion mode each character typed is inserted in the text
  - in command mode characters typed are commands to the editor

basic system administration commands: vi editor

- to change from *insertion mode* to *command mode*: press the `esc` key
- to change from *command mode* to *insertion mode*: use one of the inserting text commands
  - `i` insert characters
  - `a` append characters
  - `O,o` insert line (before of after current line)
  - ...

basic system administration commands: vi editor

x	delete a character
dw	delete a word
dd	delete a line
:w	save changes
:q	exit editor
:wq	quit saving changes
:q!	quit discarding changes

- a brief manual of the editor (although in Spanish) can be found at

`http://www.dc.fi.udc.es/~afyanez/info-vi/index.html`

Different UNIXes

Different UNIXes
→unix definitions and implementations

unix definitions and implementations

- the term *UNIX* is a vague term refering to the structure and system calls set, its origins and even sometimes to the apperarance of an OS,
- it is not a commercial brand . . . well actually it is, so only some of the *unix* systems can be branded unix. (AIX, solaris, HP/UX . . . cannot)
- the definitions impose a series of standards describing the structure, functionality, interfacing . . . .

unix definitions and implementations

- a manufacturer decides to incorporate, to a certain extent, a set of standards from a definition in the product, and then adds the characteristics (maybe from other standard) that he/she considers appropriate. Commercial interests and other issues also get in the way, giving birth to what we call an implementation.
  - example: The O.S. from Sun Microsystems was called SunOS, it was a BSD based system until version 4.1.3, the following version, SunOS 5.x switched to a System VR4 based OS, and was renamed Solaris. Solaris evolved to an open-type licence and became OpenSolaris. With the purchase of Sun by Oracle, the OpenSolaris project was dropped by the company and focused in the Solaris OS. The Opensolaris project is now somehow continued with through the OpenIndiana project

unix definitions and implementations

- There are three important standards in the unix world
  - System V
  - BSD
  - POSIX

Different UNIXes
→System V

system V

- Is the direct descendant of the ATT unix at Bell Labs
- The current standard is System V R4, which incorporates some of the characteristic of the BSD systems
- Most commercial systems are system V based (Solaris, AIX, ...)
- The most relevant System V OS these days is the Solaris OS from Oracle

Different UNIXes
→POSIX

POSIX

- POSIX stands for **P**ortable **O**pen **S**istem **I**nterface for uni**X**
- its a series of standards that defne the interface for the OS
- the most relevant OSes conforming to the POSIX standards are the linux distributions (although Solaris and some of the BSD systems also conform to the POSIX standars to some extent)
- linux is a Free Operating System with a GNU licence
- as it's free, anyone can do a linux system with a particular set of utilities: it is what we call a linux distribution (debian, fedora, gentoo, ubuntu....)

Different UNIXes
→BSD

BSD

- BSD stands for **B**erkeley **S**oftware **D**istrubution
- Present systems are free systems with a BSD licence
- The most relevant
  - freeBSD: the most extended, with more utilites and derivatives (PCBSD, DragonflyBSD ...)
  - netBSD: emphasis on portability
  - openBSD; emphasis on security