Capítol 2

Comandes, conceptes bàsics de Linux i *scripting*

Introducció

En aquest tercer tema, es pressuposa que ja s'ha instal·lat correctament el sistema operatiu. L'objectiu d'aquest capítol és que el futur administrador de sistemes es familiaritzi amb l'entorn de la línia de comandes d'un sistema operatiu Linux. Per a fer-ho, en primer lloc es presentarà el concepte de terminal (TTY), seguidament es presentaran les comandes de navegació més importants així com la organització del sistema de fitxers. Finalment, es presentaran d'altres comandes útils, les quals, amb l'ajuda dels operadors globals explicats a l'última secció d'aquest capítol, permetran dur a terme tasques d'una manera molt eficient.

Per a la lectura d'aquest capítol, seria interessant que tinguéssiu un terminal a davant per anar fent proves. Dediqueu el temps just i imprescindible per a les comandes, ja tindreu temps durant la sessió de laboratori per a provar-les amb deteniment.

2.1 Iniciar la sessió

Si la instal·lació s'ha efectuat correctament, i el gestor d'arrancada ha estat capaç d'iniciar el sistema operatiu Linux, ens apareixerà l'administrador d'inici de sessió en mode gràfic (en endavant `X Window` o `X`). En aquest capítol estem interessats en el mode orientat a comandes. Tot i que el mode gràfic té un intèrpret de comandes, és preferible entrar en una terminal de comandes.

Sovint, quan es configura un servidor, es considera una pèrdua de temps i de capacitat a disc el fet d'instal·lar tots els paquets i drivers associats a `X Window`. A la vostra pràctica, haureu de decidir en quines màquines cal instal·lar les X i en quines no.

Linux és un sistema operatiu multitasca i multiusuari, això significa que proporciona un seguit de tècniques interessants per fer varies coses simultàniament. Per exemple,

podem iniciar la compilació d'un software molt extens, seguidament canviar i descarregar el nostre correu al mateix temps que podem navegar per Internet. Actualment, això no és cap novetat, qualsevol sistema Windows permet fer-ho. El que sí que és una novetat, és que tot això és pugui fer en un entorn orientat a línia de comandes.

2.1.1 Consoles virtuals

En els seus inicis Linux era un sistema orientat a una arquitectura client-servidor: tots els clients és connectaven contra el mateix servidor, el servidor executava les comandes, i les retornava al client. Llavors, cada client és considerava que era un teletip (`TeleTYpe`). Actualment s'ha volgut seguir la mateixa filosofia, amb l'única diferència que tots els clients estan a la mateixa màquina que el servidor.

Per defecte, Linux ve amb 8 clients, o consoles virtuals, anomenades $\texttt{tty}_i$, numerades des de `tty1` fins a `tty8`. Cal saber que, per defecte la `tty7` és l'única consola virtual amb X. La resta, són consoles orientades a línia de comandes. Per canviar de terminal `tty`, només cal prémer Ctrl + Alt + F_i[1], on $_i$ indicarà el número de terminal al que volem anar.

És interessant que feu proves amb els terminals, provant de fer login en varis d'ells simultàniament. Observeu que el fet d'estar connectat en un terminal no impedeix connectar-se en un altre.

2.2 El CLI i l'arquitectura de fitxers

Com hem dit anteriorment, treballarem en un terminal de comandes, el `tty1` per exemple. En primer lloc, ens demanarà login i password, caldrà introduir els mateixos que s'han introduït durant la instal·lació. Després de superar la fase d'autenticació, per defecte, sortirà quelcom similar a això:

```
foo@joan-desktop-UB:~$
```

A continuació, analitzarem la línia anterior:

- `foo` és el nom d'usuari que acaba de fer login.
- `joan-desktop-UB` és el nom de la màquina a la que `foo` ha fet login.
- `~` significa que actualment estem al directori `home` (denotat amb el símbol `~`) de l'usuari foo.
- `$` significa que actualment no tenim permisos d'administrador, com veurem en el capítol següent, per tenir-los hauria d'haver un `#`.

[1]Compte, si s'està treballant amb VMWare caldrà configurar-lo per a que el *keystroke* de commutació host ⇔ guest no sigui el Ctrl+Alt.

2.2.1 Navegació bàsica pel sistema de fitxers

En aquesta secció, descriurem com navegar pel sistema de fitxers d'un sistema operatiu Linux. Abans de començar, cal tenir presents unes quantes de recomanacions:

1. **Els fitxers ocults, realment no estan ocults**: A diferència d'altres sistemes operatius, com Windows, en els que els fitxers ocults no poden ser vistos físicament per l'usuari, Linux tracta els fitxers ocults posant-hi un "."davant del nom del fitxer. Amb les comandes típiques de visualització de fitxers aquest tipus d'arxius no es veuen, però sempre es poden posar modificadors a les comandes per tal de mostrar-los. Exemple d'arxius ocults són el `.bashrc` o el `.bash_history`.

2. **Les extensions dels fitxers no tenen cap sentit en Linux**: Tal i com veurem més endavant, Linux utilitza el concepte *Magic Number* per tal de saber què és cada fitxer. El fitxer `AixoEsUnFitxerDeText.txt` pot ser un fitxer executable. De la mateixa manera, Linux admet noms de fitxer com `Aixo.Es.Un.Fitxer.De.Text`.

3. **Linux és un sistema operatiu *Case Sensitive***: Els fitxers `Fitxer1` i `fitxer1` són diferents. De la mateixa manera, és bo que els noms dels fitxers i directoris no tinguin accents, ni espais en blanc ni caràcters "estranys".

Per tal de navegar per les carpetes de sistema, típicament s'utilitzen tres comandes: `pwd`, `cd` i `ls` , els quals descriurem a continuació.

La comanda pwd

La comanda `pwd` retorna el *path* actual on es troba l'usuari. Així, si des de la home executem la comanda `pwd`:

```
foo@joan-desktop-UB:~$ pwd
/home/foo
```

Veiem que ens ha retornat la ruta exacta del directori home de l'usuari foo. Noteu que, en Linux, els clàssics identificadors de disc dels sistemes Microsoft (c:/) no existeixen. Linux no parla de discs, parla de punts de muntatge. Llavors, l'arrel del punt de muntatge actual és "/".

En molts casos la informació que ens donarà la comanda `pwd` i la que ens donarà el propi *prompt* serà similar. És interessant que esbrineu en quins casos passarà això.

La comanda ls

La comanda `ls` s'utilitza per llistar directoris i/o fitxers. En executar la comanda `ls`, sense cap modificador, mostrarà tots els fitxers i/o directoris del *path* actual, en columnes, sense cap més informació:

```
foo@joan-desktop-UB:~/Examples$ ls
case_Contact.pdf                    logos
case_howard_county_library.pdf      oo-about-these-files.odt
case_KRUU.pdf                       oo-about-ubuntu-ru.rtf
case_OaklandUniversity.pdf          oo-derivatives.doc
case_oxford_archaeology.pdf         oo-maxwell.odt
case_Skegness.pdf                   oo-payment-schedule.ods
case_ubuntu_johnshopkins_v2.pdf     oo-presenting-kubuntu.odp
case_ubuntu_locatrix_v1.pdf         oo-presenting-ubuntu.odp
case_Wellcome.pdf                   oo-trig.xls
fables_01_01_aesop.spx              oo-welcome.odt
gimp-ubuntu-splash.xcf              Ubuntu_Free_Culture_Showcase
kubuntu-leaflet.jpg
```

En canvi, si utilitzem el modificador `-l` i executem la comanda `ls -l` veiem la llista amb una sèrie de detalls:

```
foo@joan-desktop-UB:~/Examples$ ls -l
total 6828
 -   rw-r--r--  1   root  root  184905  2008-10-22 09:11  case_Contact.pdf
 -   rw-r--r--  1   root  root   56530  2008-10-22 09:11  case_howard_county_library.pdf
 -   rw-r--r--  1   root  root  363503  2008-10-22 09:11  case_KRUU.pdf
 -   rw-r--r--  1   root  root   57270  2008-10-22 09:11  case_OaklandUniversity.pdf
 -   rw-r--r--  1   root  root   55698  2008-10-22 09:11  case_oxford_archaeology.pdf
 -   rw-r--r--  1   root  root  133005  2008-10-22 09:11  case_Skegness.pdf

 (1)      (2)      (3)   (4)    (5)     (6)          (7)                    (8)
```

(1): Tipus d'entrada:

- **d**: directori.
- **l**: enllaç.
- **-**: fitxer.
- **b**: dispositiu d'E/S orientat a transferència de dades en bloc.
- **c**: dispositiu d'E/S orientat a transferència de dades en caràcter.

(2): Permisos.

- **r**: Permís de lectura habilitat.
- **w**: Permís d'escriptura habilitat.
- **x**: Permís d'execució habilitat.

Els 3 primers caràcters corresponen als permisos del propietari, els següents 3 caràcters corresponen als permisos de grup i els 3 últims caràcters corresponen als permisos per a la resta.
(3): Nombre d'entrades que apunten a l'entrada actual.
(4): Propietari de l'entrada.
(5): Grup al que pertany l'entrada.
(6): Mida en bytes de l'entrada.
(7): Data i hora de l'última modificació.
(8): Nom de l'entrada.

És interessant que sapigueu utilitzar altres modificadors de la comanda ls, a saber:

- `ls /proc/sys`: Mostra les entrades del directori `/proc/sys`.
- `ls -a`: Mostra les entrades incloent les ocultes (que comencen per .).
- `ls -d`: Només mostra el nom del directori, no el contingut.
- `ls -i`: Mostra la informació de l'inode de cada entrada.
- `ls -lh`: Mostra la sortida en format *human readable*. És interessant comparar-la amb la `ls -l`.

Noteu que els arguments és poden concatenar, de manera que es poden executar comandes com `ls -laih /proc`. Per a veure tots els modificadors de la comanda `ls` només cal que executeu la comanda `man ls` i navegueu amb el cursor. Per a sortir del *man*, només cal polsar la tecla 'q'.

És important entendre el significat de dues entrades importants quan un executa la comanda `ls -la`: les entrades `"."` i `".."`.

L'entrada `.` és un punter al path actual. L'entrada `..` és un punter al nivell anterior del path.

La comanda cd

La comanda `cd` per si mateixa porta a la *home* de l'usuari. Aquesta comanda és una de les més utilitzades per moure's a través de l'arbre de directoris de Linux.

Quan la comanda `cd` va acompanyada d'un argument, significa que es vol modificar el path actual i anar a la ruta indicada a l'argument. Llavors, `cd /proc/sys` canviaria el path actual al path `/proc/sys`.

Pel que fa als arguments cal entendre que hi ha dos tipus d'arguments: els que porten a un path relatiu, i els que porten a un path absolut:

- Quan l'argument o ruta, a l'inici porta el caràcter `/`, significa que és vol anar a un path absolut. Això és, a partir del punt de muntatge de la unitat. En sistemes Windows, és l'equivalent a posar `c:\`. Per exemple, `cd /proc/sys`.
- Quan l'argument o ruta, a l'inici no porta el caràcter `/`, o porta el caràcter `./`, significa que el path és relatiu. És a dir, que volem anar a un subdirectori del path actual. Per exemple, `cd sys`.

Ara estem en condicions d'acabar d'entendre les entrades "`.`"i "`..`": suposem que estem a la nostra home (podem anar-hi tot executant `cd`), executem un `pwd` per assegurarnos de que estem en el path correcte. Si fem un `cd .` i després executem `pwd`, veiem que el path no s'ha modificat. En canvi, si executem `cd ..` i després executem `pwd`, veiem que el path ha baixat un nivell.

2.2.2 L'arrel del sistema

Tal i com s'ha esmentat anteriorment, l'arrel del sistema o el punt de muntatge de la unitat física, en els sistemes operatius Linux, ve representada pel caràcter `/`. Recordem que això és l'equivalent al `c:\` dels sistemes operatius basats en Microsoft. A continuació descriurem les carpetes que pengen de l'arrel del sistema.

A la Taula 2.1 teniu els directoris més importants, que cal saber, de cara a navegar pel sistema. Aquesta jerarquia sempre hauria de ser igual en tots els sistemes Linux. Això és degut a que la disposició dels directoris en aquesta classe de sistemes, segueix l'estàndard FHS (*Filesystem Hierarchy System*), del qual podreu trobar més informació a `http://www.pathname.com/fhs/`.

2.2.3 On viuen els programes?

L'standard FHS no permet que els programes creïn els seus propis directoris en la carpeta `/usr`. Els únics directoris que s'admeten a sobre de `/usr` són:

- **bin**: Conté les comandes de l'usuari.
- **include**: Conté les capçaleres necessàries per als programes en C.
- **lib**: Conté les llibreries.

Directori	Descripció
bin	Fitxers binaris utilitzats per tots els usuaris
boot	Fitxers del Kernel, mapa del sistema i fitxers d'arrencada
dev	Fitxers que apunten als dispositius hardware del sistema
etc	Fitxers de configuració del sistema
home	Carpetes `home` de cada usuari
lib	Llibreries compartides i mòduls necessàris per compilar software
lost+found	Directori per emmagatzemar fitxers descatalogats
mnt	Directori per muntar unitats externes
opt	Directori per instal·lar software variat
proc	Informació del Kernel i dels processos del sistema
root	Carpeta `home` de l'usuari *root*
sbin	Fitxers binaris del sistema (només usables per l'usuari *root*).
tmp	Dades temporals
usr	Dades (fitxers i programes) públiques, de només lectura, disponibles per a compartir entre usuaris.
var	Dades de variables, logs, web, ftp i d'altres.

Taula 2.1: Carpetes més importants del sistema

- **local**: Conté programes locals i programes compartits. En aquest directori l'administrador de sistemes pot instal·lar software localment.
- **sbin**: Conté binaris no essencials per al sistema.
- **share**: Conté dades i/o programes per a múltiples arquitectures.

En resum, quan l'administrador del sistema ha d'instal·lar un nou software, s'ha de preguntar si vol que aquest software estigui compartit o no. Si es vol que a aquest nou software tingui accés més d'un usuari, haurà d'anar al directori `/usr/local/ElNouSoftware`, en canvi, si es vol que el software només sigui per un usuari haurà d'anar a la carpeta `/opt/ElNouSoftware`.

2.2.4 Operacions amb fitxers i directoris

Un cop ja disposem de les comandes bàsiques per poder navegar pel sistema, és moment d'il·lustrar com es poden crear i eliminar directoris en linux, així com realitzar el clàssic *cut & paste* amb Linux.

Copiar fitxers i directoris

Per tal de copiar fitxers, d'un path a un altre, és tant fàcil com utilitzar la comanda `cp PathOrigen PathDesti`. Cal tenir present que existeixen varis modificadors, molt interessants, de la comanda `cp`, a saber:

- `-d`: No segueix els enllaços, còpia l'enllaç literalment.

- `-f`: No pregunta si es vol sobreescriure, sobreescriu directament.
- `-i`: Pregunta abans de sobreescriure.
- `-l`: Crea un enllaç *hard-link* cap al fitxer origen.
- `-r` o `-R`: Atravessa subdirectoris recursivament.
- `-s`: Crea un *soft-link* cap al fitxer destí.
- `-u`: (*Update*) Només copia aquells fitxers origen que són més nous que el destí. És interessant veure què passa quan el destí no existeix.
- `-x`: No creua a altres sistemes de fitxers.

Altra vegada, tots aquests modificadors es poden concatenar. És interessant entendre el significat de la comanda `cp -rf fit1 fit2`.

Moure fitxers i directoris

Tal i com s'ha comentat anteriorment, existeix una eina capaç de fer un *cut & paste* en Linux. Aquesta és la comanda `mv PathOrigen PathDesti`. Els modificadors que admet aquesta comanda són els mateixos que els de la comanda `cp`. Compte, la comanda `mv` per defecte és recursiva, això és, ve amb el modificador `-r` per defecte.

Crear directoris

Per tal de crear un directori, cal usar la comanda `mkdir path`. Quan es vol crear una estructura de directoris donada, sovint s'utilitza el modificador `-p`. Per exemple `mkdir -p dir1/dir2/dir3/dir4`. És interessant que entengueu el funcionament d'aquest modificador.

Eliminar directoris

Per eliminar directoris, existeix la comanda `rmdir path`. De la mateixa manera que amb la comanda `mkdir`, aquesta comanda també admet el modificador `-p`.

Eliminar objectes

Un dels problemes que troba l'administrador de sistemes quan vol esborrar un directori, és que típicament el directori no està buit; en aquests casos, la comanda `rmdir` no funciona. Cal una eina que esborri el directori directament.

Existeix la comanda `rm path`, la qual esborra el *path*, ja sigui un directori o un fitxer. La potència d'aquesta comanda resideix en el fet de que admet els mateixos modificadors que la comanda `cp`. Llavors, si es vol esborrar un directori, sense preocupar-se de si el directori està buit o no, es pot fer el clàssic `rm -rf directori`. Cal anar amb compte amb el que s'esborra amb aquesta comanda.

2.3 Comandes interessants

2.3.1 La comanda stat

La comanda `stat` mostra informació sobre el fitxer o sistema de fitxers que rep com a paràmetre.
Per exemple, executem stat sobre un fitxer anomenat *'fitxer1'*:

```
is14104@cygnus:\~{}/comandes> stat fitxer1
File: ‘fitxer1’
Size: 0            Blocks: 0    IO Block: 32768  regular empty file
Device: 12h/18d Inode: 598053      Links: 1
Access: (0600/-rw-------)  Uid: (10201/ is14104)   Gid: (10000/   IS)
Access: 2009-10-01 20:18:18.000000000 +0200
Modify: 2009-10-01 20:18:18.000062000 +0200
Change: 2009-10-01 20:18:18.000062000 +0200
```

Taula 2.2: Sortida de la comanda `stat fitxer1`.

Es pot apreciar la diferent informació sobre el fitxer, tal com el nom del mateix, la mida, els blocs que ocupa, el tipus de fitxer, el tipus de permisos, l'usuari propietari del fitxer, el grup propietari del fitxer i les diferents dates del darrer accés, de modificació i de canvi d'estat del fitxer.

Cal tenir present que:

- Al sobreescriure un fitxer es modifiquen els temps d'**accés**, de **modificació** i de **canvi d'estat**.
- Al canviar els permisos o propietari d'un fitxer es modifiquen el temps de **canvi d'estat** i el temps d'**accés**.
- Al llegir un fitxer es modifica el temps d'**accés**.

2.3.2 La comanda file

Degut a que en un sistema operatiu Linux les extensions perden el sentit físic, cal una eina que mostri de quin tipus és cada fitxer. Aquesta eina és la comanda `file`. Per exemple, `file /bin/ls`, dóna una sortida interessant.

2.3.3 La comanda touch

Hi ha varies raons per utilitzar la comanda `touch`. Típicament, `touch` *`NomFitxer`* poden passar dues coses:

- Si el fitxer no existeix, crearà un fitxer, en el path indicat per *`NouFitxer`*.

- Si el fitxer existeix, actualitzarà les tres dates que s'emmagatzemen d'un fitxer: la de creació, la de canvi i la de modificació. És interessant que ho comproveu amb la comanda `stat`.

2.3.4 La comanda diff

Aquesta comanda serveix per comparar dos fitxers a nivell de línia; és molt utilitzada per generar *patches* de *software*. És molt important que experimenteu amb aquesta comanda i els seus modificadors per tal d'entendre la seva utilitat.

A continuació es mostra la sortida que treu aquesta comanda (extreta d'un dels *patches* que heu d'aplicar a la FASE1 de la pràctica). És molt important entendre el significat de cada línia.

```
diff -Naur perl-5.10.1.orig/hints/linux.sh perl-5.10.1/hints/linux.sh
--- perl-5.10.1.orig/hints/linux.sh     2009-02-12 22:58:12.000000000 +0000
+++ perl-5.10.1/hints/linux.sh  2009-08-25 18:31:06.000000000 +0000
@@ -63,9 +63,9 @@
 # We don't use __GLIBC__ and  __GLIBC_MINOR__ because they
 # are insufficiently precise to distinguish things like
 # libc-2.0.6 and libc-2.0.7.
-if test -L /lib/libc.so.6; then
+if test -L ${prefix}/lib/libc.so.6; then
     libc=`ls -l /lib/libc.so.6 | awk '{print $NF}'`
-    libc=/lib/$libc
+    libc=${prefix}/lib/$libc
 fi

 # Configure may fail to find lstat() since it's a static/inline
@@ -436,3 +436,8 @@
     libswanted="$libswanted pthread"
     ;;
 esac
+
+locincpth=""
+loclibpth=""
+glibpth="${prefix}/lib"
+usrinc="${prefix}/include"
```

2.3.5 La comanda man

La comanda man és una manera ràpida de buscar la sintaxis i el funcionament de qualsevol comanda en un sistema operatiu Linux. És tant senzill com escriure `man comanda`. Llavors, amb els cursors es pot anar navegant pel text. Per sortir del manual, només

cal prémer la tecla 'q'.

A través del `man`, seria interessant que busquéssiu la sintaxis i provéssiu el funcionament de les següents comandes:

```
cat
more
less
tail
grep
apropos
head
sort
cal
date
ps
tree
tac
```

2.4 Com cercar fitxers

La manera més ràpida de cercar fitxers en un sistema operatiu Linux és mitjançant la comanda `locate`. És l'eina més ràpida i eficient del sistema per fer aquesta tasca. Això és degut a que, quan realitza una cerca, no treballa directament contra els continguts reals del disc, sinó que treballa contra una base de dades.

La sintaxis d'aquesta comanda és `locate PatternACercar`. Cal anar amb compte, la comanda locate examina *tots* els paths del sistema i escriu per pantalla els que continguin el `PatternACercar`, ja sigui al final del path, al principi, o entre mig. És interessant que proveu de fer:

```
mkdir -p llegeix/llegei
locate llegei
```

Veureu, que el sistema no troba cap coincidència. Això és degut a que encara no hem actualitzat la base de dades que la comanda `locate` utilitza per fer la cerca. Per actualitzar la base de dades, n'hi ha prou en executar una d'aquestes dues comandes:

```
sudo updatedb
sudo slocate -u
```

És lògic pensar que l'actualització de la base de dades s'haurà de realitzar d'una manera periòdica. Serà feina de l'administrador de sistemes programar aquesta tasca per a que

s'executi a voluntat.

Per tal d'incrementar el temps de cerca, i actualització, `locate` llegeix el fitxer `/etc/updatedb.conf`, en el que típicament es pot trobar quelcom similar a això:

```
PRUNE_BIND_MOUNTS="yes"
PRUNEPATHS="/tmp /var/spool /media"
PRUNEFS="NFS nfs nfs4 rpc_pipefs afs binfmt_misc proc smbfs autofs iso9660
 ncpfs coda devpts ftpfs devfs mfs shfs sysfs cifs lustre_lite tmpfs usbfs
 udf"
```

En aquest fitxer, s'indiquen a la comanda `locate` dues coses:

- `PRUNEPATHS`: *Paths* en els que la comanda `locate` no cercarà.
- `PRUNEFS`: Sistemes de fitxers en els que la comanda `locate` no cercarà.

2.5 Els modificadors globals

Entrada i sortida estàndard i combinació de comandes

Entendrem per entrada estàndard el dispositiu per on l'usuari introdueix dades al sistema. Així, Des de l'entrada estàndard els diferents programes obtindran les dades que necessiten. Per defecte l'entrada estàndard és el teclat. Per altra banda, la sortida estàndard dirigeix la sortida que l'usuari obté i per defecte és la pantalla del terminal.

En Linux podem redirigir tant l'entrada com la sortida. Això implica que el shell[2] canviï a on envia la sortida o d'on recull l'entrada de dades. Per a fer-ho caldrà reconfigurar quina és la sortida estàndard (abans la pantalla del terminal) i quina l'entrada estàndard (abans el teclat). Per a fer-ho utilitzarem els operadors $<$, $>$ i $>>$:

- Per a redirigir l'entrada estàndard s'utilitzarà l'operador $<$. Per exemple, si volem redirigir un fitxer cap a la comanda `cat` farem `cat < fitxer`. Tot i que no és necessari en aquest cas particular, ens podem trobar situacions on calgui redirigir l'entrada a un fitxer.
- Per redirigir la sortida estàndard s'utilitzarà l'operador $>$, si volem sobreescriure la sortida i l'operador $>>$ si volem afegir al que ja tingui la sortida. És important tenir en compte la diferència entre els dos operadors. Aquesta diferència la veuren en l'exemple següent.

Per exemple, si volem guardar un llistat dels fitxers que tenim al directori `/space`, tenint en compte que el directori conté els fitxers `cassini, moons, pioneer, planets` i `voyager` farem:

```
ls -l >  continguts.txt
```

[2]La *shell* és l'intèrpret de comandes del sistema.

redirigirà la sortida de la comanda `ls -l` cap al fitxer `continguts.txt`, **sobreescrivint** el contingut d'aquest. Si revisem el fitxer `continguts.txt` ens donarà la següent sortida:

```
total 20
-rw-r--r-- 1 cooller cooller 16 2009-10-02 14:24 cassini
-rw-r--r-- 1 cooller cooller  0 2009-10-02 14:27 continguts.txt
-rw-r--r-- 1 cooller cooller 33 2009-10-02 14:26 moons
-rw-r--r-- 1 cooller cooller 16 2009-10-02 14:25 pioneer
-rw-r--r-- 1 cooller cooller 30 2009-10-02 14:26 planets
-rw-r--r-- 1 cooller cooller 16 2009-10-02 14:27 voyager
```

Si ara volem guardar la ruta del directori, fent `pwd` i redirigint cap al mateix fitxer

```
pwd >> continguts.txt
```

redirigirà la sortida de la comanda `pwd` cap al fitxer `continguts.txt`, però afegirà el resultat al final del fitxer, **sense** sobreescriure'l. Així la sortida serà:

```
total 20
-rw-r--r-- 1 cooller cooller 16 2009-10-02 14:24 cassini
-rw-r--r-- 1 cooller cooller  0 2009-10-02 14:27 continguts.txt
-rw-r--r-- 1 cooller cooller 33 2009-10-02 14:26 moons
-rw-r--r-- 1 cooller cooller 16 2009-10-02 14:25 pioneer
-rw-r--r-- 1 cooller cooller 30 2009-10-02 14:26 planets
-rw-r--r-- 1 cooller cooller 16 2009-10-02 14:27 voyager
/home/cooller/space
```

Fixeu-vos com s'ha afegit la ruta al directori en l'última línia. Què hagués passat si haguéssim emprat l'operador > en comptes de >> en la comanda `pwd`? La resposta es deixa pel lector.

En els sistemes Linux, existeix un fitxer bastant particular, el qual és comporta com un forat negre: tot el que vagi cap allà, entrarà, però no sortirà mai. És bastant curiós executa la comanda `cat ls -lai > /dev/null` i després fer un `cat /dev/null` i comprovar que el fitxer `/dev/null` segueix buit. Seria interessant esbrinar per què passa això.

Podem connectar la sortida estàndard d'una comanda amb l'entrada d'una altra. Per a fer això emprarem una *pipe*. Aquesta la representarem amb el símbol `|` i anirà entre dues comandes. Per entendre-ho millor, fixem-nos en el següent exemple:

```
ls -l /usr/include | more
```

Concatenarà la sortida de la comanda `ls -l` del directori `/usr/include` i mostrarà de forma paginada aquesta. També podem executar de forma seqüencial diverses comandes. Això pot resultar interessant en certes ocasions. Per a fer-ho escriurem les diverses comandes separades per `;`. Per exemple:

```
date; sort planets > ordenat.txt; date
```

D'aquesta manera obtindrem el temps que triga a ordenar el contingut del fitxer `planets`.

2.6 Scripts shell I. Introducció

Després de familiaritzar-se amb el sistema operatiu Linux, és moment d'entrar en el món de l'scripting. Aquest és un tema en el que temari oficial LPI no es dóna una especial importància. Afortunadament, s'ha cregut que, segons les demandes actuals del mercat, un bon administrador de sistemes ha de ser capaç de dissenyar, programar i depurar scripts.

De cara al temari global de l'assignatura, s'ha decidit separar la secció d'scripts en dues parts: una primera part de nivell elemental (la qual està constituida per aquest capítol) i una segona part de nivell avançat (la qual està constituida per bona part del capítol següent).

És evident que els scripts no són estrictament imprescindibles, ja que tal i com es veurà al llarg d'aquesta secció, no són més que seqüències d'instruccions que l'administrador pot anar executant una rere l'altra. No obstant, un administrador que treballi amb scripts, serà capaç de realitzar la seva feina eficaçment i el que és més important, eficientment.

La secció s'estructura de la següent manera: en primer lloc es presenten els scripts, seguidament es descriu el seu funcionament mitjançant una sèrie d'exemples simples, a continuació es descriuen les sentències més importants i finalment es conclou el capítol amb l'eina awk.

Noti's que aquest és un capítol eminentment pràctic, pel que en aquest cas, més que mai, és altament recomanable anar provant tots els scripts i comandes que es van presentant.

2.6.1 Els scripts shell

Un script shell (en anglès *shell script*) són un seguit de comandes i crides que l'intèrpret de comandes del sistema operatiu s'encarregarà d'executar seqüencialment. D'aquesta manera es poden construir programes emprant comandes Linux/Unix, executables propis, etc, utilitzant un o diversos fitxers. A part també ens ofereix una sèrie d'estructures de control similars a les dels llenguatges de programació clàssics, amb l'avantatge de no s'han de compilar[3].

Un bon administrador de sistemes ha de saber quan és necessari l'ús d'scripts i quan no ho és. En general, emprarem scripts per automatitzar tasques de tot tipus o, si més no, per simplificar-les enormement. El coneixement dels scripts shell és primordial per a qualsevol administrador de sistemes.

Tal i com es desprén del paràgraf anterior, hi haurà situacions en les que no convindrà emprar scripts shell i caldrà plantejar-se el fet d'utilitzar un llenguatge de programació

[3]Els scripts s'interpreten, no es compilen. Això vol dir que serà necessari tenir un intèrpret, en el nostre cas serà *bash*.

tradicional (com és el cas de `C, C++`, etc.). A continuació es presenten una sèrie de pautes per tal de guiar a l'administrador davant del dilema d'utilitzar, o no, scripts. En general, **no** s'utilitzaran scripts shell:

- En aquelles tasques que consumeixen molts recursos i, especialment, en aquelles on la velocitat sigui un factor clau.
- En aplicacions complexes on es requereixi d'un llenguatge d'alt nivell, per exemple orientat a objectes.
- En tasques amb molts accessos a fitxers, doncs els scripts shell només permeten treballar en format seqüencial i de forma poc eficient.
- Quan hi hagi la necessitat d'utilitzar una interfície d'usuari o de manipular fitxers multimèdia.
- Quan calgui accedir directament al hardware de la màquina.
- Quan calgui treballar amb accessos a xarxa, com per exemple l'ús de *sockets*.

2.6.2 Programació amb scripts shell: *bash*

Un script constarà obligatòriament d'una primera línia que començarà amb els caràcters `#!` (això és l'anomenat *magic number* el qual indica al sistema operatiu que es tracta d'un script) juntament amb el path de l'intèrpret que s'utilitzarà. En el nostre cas, sempre utilitzarem l'intèrpret *bash*. Llavors, en la primera línia dels nostres scripts tindrem:

```
#!/bin/bash
```

que tal i com hem dit, indicarà que estem utilitzant un script, el qual ha de ser interpretat per bash. A continuació, es presenta un exemple molt simple d'un script:

```
#!/bin/bash
    # Aquest és el meu primer script
echo "Hola, món!"
```

Cal notar que les línies que comencen per `#`, a excepció de la del *magic number*. Recordeu que els comentaris són importants a l'hora de programar en qualsevol llenguatge.

Després d'entendre conceptualment què és un script, és moment de preguntar-se com s'executa un script. Primerament caldrà guardar el fitxer a disc mitjançant qualsevol dels editors vistos (vi, vim, nano, pico, etc...). Com a regla general, els scripts tindran extensió `.sh` tot i que això no és obligatori, ens facilitarà enormement el fet d'identificar-los. Un cop guardat a disc, podem passar a executar-lo. Per a fer-ho tenim dues opcions, suposat que hem anomenat al fitxer `hello.sh`:

1. Invocar el intèrpret i passar-li com a argument l'script. En el nostre cas faríem

```
bash hello.sh
```

2. Dotar de permisos d'execució a l'script i executar-lo com si fos un binari tradicional:

```
chmod +x hello.sh
./hello.sh
```

L'exit status

En aquest apartat es descriurà el funcionament de l'*exit status*. L'*exit status* permet indicar com a finalitzat un programa. Evidentment, podrem forçar aquest valor: n'hi haurà prou en utilitzar la comanda `exit`. Aquesta comanda finalitzarà l'script o el programa, retornant un valor: (l'*exit status*). Aquest valor podrà ser llegit pel shell des del qual es va executar. Si el valor de l'*exit status* és 0 significarà que l'execució del programa ha resultat satisfactòria, evidentment, en cas de produir-se algun error serà diferent de 0.

Això és el que passa amb un programa fet en C, on al finalitzar la funció *main* retornem un 0. Aquest valor resultarà molt útil per tal de monitoritzar tasques de forma automàtica.

Per saber l'exit status de l'última comanda o script shell executat, existeix la variable d'entorn `$?`. Anem a veure-ho amb un exemple. Suposem que anomenem al següent script `es1.sh`:

```
#!/bin/bash
#farem un ls normal i mirarem el seu exit status
ls
echo "el valor de l'exit status de ls és $?"
```

El que fa aquest script és el següent: llença un `ls` (noteu que el resultat de la comanda `ls` es mostra per pantalla) i finalment la comanda `echo` ens mostra la frase. Si executem l'script veurem que mostra que l'exit status és 0 (`ls` finalitzat correctament).

Ara veurem què passa quan llencem una comanda incorrecta: al següent script l'anomenarem `es2.sh`:

```
#!/bin/bash
#farem una comanda inexistent i mirarem el seu exit status
jfdskljfkldsjf
echo "el valor de l'exit status de jfdskljfkldsjf és $?"
```

Observem que l'script mostrarà el següent missatge:

```
es2.sh: línea 3: jfdskljfkldsjf: no se encontró la orden
el valor de l'exit status de ls és 127
```

Fixem-nos que l'exit status ha canviat: al ser una comanda incorrecta torna un valor diferent de 0 (en aquest cas el valor 127).

Ara veurem com retornar un valor d'*exit status* diferent en el nostre script emprant la comanda `exit`. De totes maneres, es recomana seguir l'estàndard i retornar 0 si el programa ha finalitzat correctament i qualsevol altre en cas d'error. Al següent script l'anomenarem `es3.sh`:

```
#!/bin/bash
echo "forcem la sortida a 22"
exit 22
```

Si volem veure l'*exit status*, des de la línia de comandes, farem:

```
    ./es3.sh
echo $?
```

I ens mostrarà el valor 22 per pantalla.

Variables en bash

Les variables en bash tenen una sintaxi especial. Les variables no tenen tipus (enter, real, booleà, cadena de caràcters), així que l'intèrpret ho assigna de forma automàtica. Quan la variable en qüestió estigui en la part esquerra d'una expressió, això és, quan assignem un valor a la variable, escriurem el nom d'aquesta, com és el cas del llenguatge C. Mirem l'exemple següent:

```
lamevavariable=33 #atenció, NO POT HAVER ESPAIS EN BLANC!
```

Fixeu-vos que entre el nom de la variable (això és, `lamevavariable`), el símbol igual i el valor **no hi ha espais en blanc**. A diferència de C, on els espais en blanc no importen, en bash s'haurà d'anar amb molt de compte amb aquests detalls.

Quan vulguem consultar el valor de la variable emprarem el símbol $ davant del nom de la variable (compte que la sintaxi ara difereix de la de C!). Per exemple, si volem mostrar el valor de la variable `lamevavariable` farem el següent:

```
echo $lamevavariable
```

Això serà cert sempre que la variable estigui en la part dreta d'una expressió (això és, que sigui consultada per a fer un càlcul) o, en general, quan consultem el seu valor.

Cal que pareu atenció amb el fet de que si una variable està dins de cometes simples (el caràcter ' (apòstrof)) no s'interpretarà el valor d'aquesta, és a dir, si tenim això:

```
echo '$lamevavariable'
```

Mostrarà per pantalla la cadena `$lamevavariable` (i no el valor de la variable). En canvi, si tenim la variable dins de comentes dobles (el caràcter ") sí que s'interpretarà el valor. És a dir, si fem el següent:

```
echo "$lamevavariable"
```

Mostrarà per pantalla el valor de la variable. És interessant que proveu el següent script, que l'anomenarem `var1.sh`:

```
#!/bin/bash
variable=33 #assignem un valor
echo $variable #mostra 33
echo '$variable' #mostra $variable
echo "$variable" #mostra 33
```

També podem assignar una cadena de caràcters a una variable, tal i com mostra l'exemple següent:

```
#!/bin/bash
v="Hola, món"
echo $v
```

I, de forma anàloga, també podem assignar el contingut d'una variable a una altra variable:

```
#!/bin/bash
v=1337
z=$v
echo $z
```

Existeixen una sèrie de variables especials. Aquestes permetran obtenir informació útil de cara a controlar diferents tasques dins de l'script. Passem a numerar-les:

- `$0, $1, $2, ...`—Aquestes variables contenen els arguments passats per línia de comandes al nostre script. És el mateix que el paràmetre `argc` de la funció `main` en el llenguatge C. Tal com passa amb C, `$0` (l'argument 0) és el nom de l'script.
- `$#`—Aquesta variable conté el número de variables que han estat passats a l'script. És el mateix que el paràmetre `argc` de la funció `main` en el llenguatge C.
- `$$`—PID corresponent al nostre script.
- `$*`—Cadena que conté tots els arguments passats al nostre script.

Per tal de veure millor com funciona tot plegat farem un exemple pràctic. Aquest script mostrarà els dos primers arguments (si en té), juntament amb la resta d'informació. A aquest script l'anomenarem `var2.sh`:

```
#!/bin/bash
echo "Paràmetre 0: $0" #sempre conté el nom de l'script!
echo "Paràmetre 1 (si en te): $1"
echo "Paràmetre 2 (si en te): $2"
echo "Número de paràmetres totals: $#"
echo "PID de l'script: $$"
echo "Tots els paràmetres: $*"
```

Observem què passa quan executem aquest script amb diferents paràmetres. Primerament passem dos paràmetres, anomenats `parametre1` i `parametre2` respectivament:

```
assoo@assoo-desktop:~/scripts$ bash var2.sh parametre1 parametre2
Paràmetre 0: var2.sh
Paràmetre 1 (si en te): parametre1
Paràmetre 2 (si en te): parametre2
Número de paràmetres totals: 2
PID de l'script: 16362
Tots els paràmetres: parametre1 parametre2
```

Fixem-nos que el número total de paràmetres és 2. Provem ara amb cinc paràmetres:

```
assoo@assoo-desktop:~/scripts$ bash var2.sh A B C D E
Paràmetre 0: var2.sh
Paràmetre 1 (si en te): A
Paràmetre 2 (si en te): B
Número de paràmetres totals: 5
PID de l'script: 16364
Tots els paràmetres: A B C D E
```

Ara provem que passa si no passem cap paràmetre:

```
assoo@assoo-desktop:~/scripts$ bash var2.sh
Paràmetre 0: var2.sh
Paràmetre 1 (si en te):
Paràmetre 2 (si en te):
Número de paràmetres totals: 0
PID de l'script: 16365
Tots els paràmetres:
```

Variables i cadenes de caràcters

Les cadenes de caràcters en bash poden estar, com hem vist anteriorment, tant entre cometes dobles com entre cometes simples (la diferència, recordem, estava en que les primeres interpreten les variables i les segones no). Podem, igual que ocorre amb C, inserir caràcters especials com tabulacions amb `\t` o salts de línia amb `\n`. Però atenció, per tal d'indicar que volem que s'interpretin aquests caràcters especials haurem de passar-li l'opció `-e` a la comanda `echo`. Per veure-ho millor fixem-nos amb el següent script, que anomenarem `var3.sh`:

```
#!/bin/bash
echo -e "Abans de tabular \t Després de tabular"
echo -e "Farem salts de línia \n un salt\n dos salts"
```

La sortida del qual és la següent:

```
assoo@assoo-desktop:~/scripts$ bash var3.sh
Abans de tabular         Després de tabular
Farem salts de línia
 un salt
 dos salts
```

Sense el paràmetre `-e` la comanda `echo` no interpretarà aquests caràcters especials. És interessant que es faci la prova traient el paràmetre de l'`echo`.

Si una cadena de caràcters està entre accents oberts (cometes invertides, el caràcter ‘), el valor d'aquesta serà el resultat de l'execució de les comandes que contingui. Això que sembla, aparentment complicat, serà d'extrema utilitat quan treballem amb scripts més avançats. Per entendre-ho millor anem a veure un exemple, que anomenarem `var4.sh`:

```
#!/bin/bash
echo ‘ls -l‘
```

Aquest script executarà la comanda `ls -l` i la mostrarà per pantalla. Proveu de fer-ho.

Anem a veure la utilitat bàsica de posar cadenes entre accents oberts: la d'assignar valors que resulten de comandes a les variables. Al següent script l'anomenarem `var5.sh`:

```
#!/bin/bash
v1=‘whoami‘
v2=‘ps axu | grep $v1‘
echo "La variable val: $v2"
```

Tenim dues variables, `v1` i `v2`. La primera d'elles rep el valor de la comanda `whoami` la qual obté el login de l'usuari que executa la comanda. La segona variable, `v2`, realitza un `ps axu` i filtra, amb la comanda `grep`, de tal manera que només capturarà aquells usuaris indicats per la variable `v1`, o el que és el mateix, només mostrarà les línies on apareix l'usuari executor de l'script.

Per finalitzar la secció de les variables només ens queda veure com llegir de teclat. En C tenim la funció `scanf`, entre d'altres. En bash resultarà molt més senzill, utilitzarem la comanda `read`. Per veure el funcionament analitzem el següent script:

```
#!/bin/bash
echo "Entra valor:"
read v
echo "Has indroduit $v"
```

Fixem-nos que no cal definir les variables abans d'utilitzar-les; simplement les posem allà on les necessitem. Tampoc indiquem el tipus. En l'exemple anterior podem introduir tant un valor enter com una cadena de caràcters.

AWK

Aquesta última secció la dedicarem a l'awk. Awk és un llenguatge de programació molt potent que s'utilitza per processar textos des de la línia de comandes. Tot i que l'awk donaría per una sessió sencera, en aquest primer capítol introductori d'scripts es descriurà l'awk breument per tal de poder completar els scripts en bash. Per últim, si un revisa la literatura, se n'adona que d'awk s'han editat llibres sencers, alguns d'ells molt extensos!

Un programa awk resulta en una sèrie de parelles patró-acció, de la forma:

```
patró { acció }
```

On el patró és una expressió, com per exemple el fet de que una columna d'un determinat fitxer sigui més gran que un determinat valor, i l'acció contindrà les comandes que seran executades. La millor manera d'aprendre awk és a base d'exemples. Trobareu la referència tant en les pàgines del `man` com en el web `http://awk.info/`.

Imaginem que tenim un fitxer anomenat `dades.txt` amb la següent informació:

```
maria   60     1
josep   50     0
andreu  657    20
lluis   4      1
joan    525    143
```

On el significat d'aquestes dades es descriu a continuació:

- **Columna 1**: logins dels diferents usuaris del sistema.
- **Columna 2**: nombre d'hores que porten acumulades fins ara en el sistema.
- **Columna 3**: nombre d'emails enviats fins al moment.

El que es planteja ara és com es podria imprimir per pantalla els usuaris que porten més de 100 hores en el sistema. Podríem plantejar un script molt complex amb tota una sèrie de condicionals i tractament de cadenes via grep, però per sort awk ens facilita moltíssim la feina:

```
awk ' $2 > 100 { print $1 }' dades.txt
```

Aquest script té com a patró totes aquelles segones columnes (les hores acumulades) majors que 100 i com a acció pintar per pantalla la primera columna (el login de l'usuari).

De la mateixa manera, awk també permet manipular el resultat d'altres comandes emprant `pipes`, tal i com s'il·lustra en el següent exemple:

```
ls -l | awk '{print "Pertany a:",$4 }'
```

La sortida serà similar a la següent:

```
assoo@assoo-desktop:~/scripts$ ls -l | grep rwx | awk '{print "Pertany a:",$4}'
Pertany a: root
Pertany a: root
Pertany a: root
```

Aquest script mostra el grup on pertanyen tots els fitxers i carpetes del directori actual que tenen permisos de lectura, escriptura i execució.