# Permisos en Linux

En este documento pretendo describir la funcionalidad del sistema de permisos de Linux, a un nivel básico claro, no queremos enfrentarnos a un sistema tan complejo como PAM de momento. Para realizar esta introducción primero dejaremos claros los elementos involucrados en el control de permisos, las estructuras de datos utilizadas y cómo el sistema operativo utiliza esos componentes para permitir o denegar la realización de acciones en el sistema.

*INDICE*

# 1 USUARIOS – PROCESOS - FICHEROS

Existen tres conceptos básicos que es necesario tener claro antes de comenzar hablar del sistema de permisos de Linux, conceptos que además son comunes a casi todos los sistemas operativos tipo UNIX. Estos conceptos son complementarios y en algunas ocasiones la misma palabra tiene diferentes significados según se aplique a cada uno de ellos.

Estos tres conceptos son los usuarios, los procesos y los ficheros.

## 1.1 USUARIOS

Cuando hablamos de usuarios en un sistema Unix estamos hablando de la definición que se ha realizado previamente de perfiles de ejecución. Estos perfiles de ejecución pueden estar ligados a un usuario concreto o a una forma de trabajar.

Es cierto que lo más normal es la primera interpretación y que normalmente se confunde el término usuario con "cuenta de acceso" a la máquina, pero un usuario de Linux es bastante más que eso. De hecho nada más instalar Linux podemos ver que existen múltiples usuarios definidos, aunque en el proceso de instalación se nos haya preguntado y sólo hayamos creado uno o dos (contando el superusuario y root).

Esto se hace así para definir perfiles de ejecución para algunas aplicaciones que no requieran todo los que una aplicación asociada a root puede llegar a hacer en el sistema. Un ejemplo de esto es el caso de Android, donde se crea un usuario por cada aplicación instalada para poder definir mejor los permisos que tendrá dicha aplicación durante su ejecución… y eso aunque la persona que está usando el dispositivo Android sea el mismo "usuario" (desde otro punto de vista claro).

No vamos a entrar aquí en definir las múltiples formas de definir usuarios en una máquina Unix (local, NDIS, YP, ActiveDirectory….), eso se sale del propósito de este documento.

Todos los usuarios están identificados por un número y una cadena. Al número se le denomina Identificador de Usuario o UID y la cadena se la denomina "login name". Estos dos componentes son ÚNICOS para cada usuario. El sistema utiliza el UID, pero para nosotros suele ser más cómodo recordar y utilizar el "login", de ahí su existencia.

Además de esta identificación personal, a cada usuario se le asignan una serie de identificadores adicionales que se corresponden a los *GRUPOS* a los que el usuario pertenece. Estos grupos permiten definir acciones comunes para conjuntos de usuarios. Por ejemplo, puede existir el grupo conf_web que permita a todos los usuarios que pertenezcan a ese grupo acceder y modificar la configuración de un posible servidor web.

**¿Cómo saber a que grupos pertenezco?**

La orden básica para obtener información de un usuario es ***id***.

Leer la página de manual de esta orden es una buena forma de comenzar a obtener información sobre todo este sistema. Existen otras ordenes parecidas, por ejemplos groups, que permiten obtener la misma información de otra forma, como casi siempre en UNIX, hay múltipels formas de llegar al mismo resultado.

Todos los elementos del sistema tienen que pertenecer a un usuario, es decir, todos los demás elementos que nos vamos a encontrar tendrán un campo UID que indicará que usuario es el responsable/propietario de dicho elemento.

## 1.2 PROCESOS

Los procesos son programas en ejecución las aplicaciones que estamos ejecutando, hasta aquí espero que todos estemos de acuerdo. Como ya hemos comentado en el curso, los procesos en UNIX forman un árbol desde el proceso inicial y la relación de los elementos de ese árbol es de padres-hijos.

Cada proceso está etiquetado con un identificador único: su PID. Además almacena el identificador del proceso que lo creó (PPID). Si el proceso que lo creó muere este valor se pone a "1" y se considera que el proceso es hijo de INIT (proceso inicial).

Los procesos guardan información sobre el usuario que es responsable del proceso, recordemos que llamamos UID a un entero que identifica a un usuario. Pero, para complicar un poco las cosas, los procesos almacenan 4 UID y 4 identificadores de grupo. Estos identificadores se denominan:

- **UIDreal**: identificador del usuario que es el responsable de este proceso. Es un valor que no cambia a lo largo de la vida del proceso, salvo que el proceso tenga privilegios especiales.
- **GIDreal**: identificador del grupo principal al que se asocia este proceso. Normalmente es el grupo principal del usuario, pero puede cambiar bajo determinados supuestos.
- **UIDefectivo**: identificador de usuario que se utilizará para comprobar los permisos que tiene el proceso. Este valor puede cambiar a lo largo de la vida de un proceso siguiendo un conjunto de reglas que explicaremos a continuación.
- **GIDefectivo**: identificador de grupo adicional que se usará para determinar los permisos.
- **UIDsaved,GIDsaved**: UID y GID salvados del proceso. Se pueden utilizar para modificar los otros UID /GID a su valor.
- **UIDfs, GIDfs**: caso especial de LINUX. Se añadieron para poder implementar algunos servicios de Red como NFS y se han quedado. NO se suelen utilizar y toda la documentación hablará de UID y GID reales y efectivos.

Un proceso siempre puede cambiar intercambiar el valor de los UID real, efectivo y salvado.

Un proceso recién arrancado tiene en estos campos los mismos valores que el proceso que lo crea.

**Sobre los grupos**

Además del GID efectivo, un proeso también está etiquetado con un conjunto de grupos que normalmente es igual a los grupos a los que pertenece el usuario que lo creó. Esto debe estar muy presente a la hora de suponer que se va a denegar un acción a un proceso porque el GIDEfectivo no es el apropiado, a que es posible que alguno de los grupos

Cojamos el el UID, si nos preguntamos por qué es necesario tener más de un identificador el motivo es poder dar temporalmente más o menos accesos a un proceso y que el proceso siempre pueda volver a utilizar el identificador original que tenía. Es decir, si un proceso cambia su UID efectivo por lo que sea el proceso SIEMPRE podrá ver que identificador tenía originalmente y actuar en consecuencia, incluso puede volver a poner como efectivo el identificador real o el salvado, o cambiar el real por el salvado.

**¿Cómo saber los identificadores del proceso N en LINUX?**

Suponemos que N es el PID del proceso.

La forma más rápida de conocer todoa su información en LINUX es acceder a /proc/N/status. Para realizar este acceso bast con escribir en una consola: ***cat /proc/N/status***

De toda la información mostrada nos interesarán las líneas que empienzan con UID, Gid y Groups.

Además de:

*http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html*

*http://man7.org/linux/man-pages/man5/proc.5.html*

Para más información sugiero leer las páginas de manual ***setuid*** y ***seteuid y setresuid***.

## 1.3 FICHEROS

***Todo en UNIX es un fichero...***

Es bastante habitual oír esta expresión que, aunque no es completamente cierta, sí que define la filosofía de UNIX muy bien. La idea es que todo lo que no pueda recibir un nombre concreto, y hasta ahora sólo tenemos procesos y usuarios, lo identificaremos como FICHERO. Esto quiere decir que los dispositivos, la memoria, los elementos de control, las redes, las conexiones de red o los directorios se verán como elementos del sistema de fichero de UNIX.

El sistema de ficheros de UNIX es un árbol, a diferencia de Windows donde encontramos lo que yo denomino “un bosque”. Una ruta en Windows tiene el formato “**NOM_DISP:/RUTA**” donde **NOM_DISP** suele ser una letra para aquellos dispositivos que se pueden acceder como un disco. En cambio en UNIX

tenemos un solo directorio raíz "/", del que cuelgan TODOS los ficheros accesibles en un momento dado por el sistema. Como se introducen los discos y otros dispositivos de bloques en esta estructura cae fuera del objetivo de este documento, aunque siempre se puede leer la página del manual de la orden ***mount***.

Ya sabemos que las órdenes básicas para movernos por el sistema de directorios son: **cd** para cambiar de directorio, **pwd** para averiguar la ruta desde "/" del directorio donde nos encontramos y **ls** para mostrar el contenido de un directorio. Para **ls** recordar la opción **–l** si se quiere obtener información adicional sobre las entradas.

Por ejemplo, en un sistema se ha realizado "**ls –l /**" y se ha obtenido la siguiente salida:

```
mimateo@shell-labs:~$ ls -l /
total 120
drwxr-xr-x   2 root root  4096 dic  1 08:27 bin
drwxr-xr-x   3 root root  4096 dic  1 09:23 boot
drwxr-xr-x   2 root root  4096 jun 27  2011 cdrom
drwxr-xr-x  14 root root  3980 dic  3 13:09 dev
drwxr-xr-x 142 root root 12288 dic  3 13:10 etc
drwxr-xr-x   7 root root  4096 oct 10 12:12 home
lrwxrwxrwx   1 root root    33 dic  1 08:34 initrd.img -> /boot/initrd.img-3.2.0-72
drwxr-xr-x  25 root root 12288 sep 10 09:17 lib
drwxr-xr-x   2 root root  4096 sep 10 09:17 lib32
drwxr-xr-x   2 root root  4096 sep 10 09:17 lib64
drwx------   2 root root 16384 jun 27  2011 lost+found
drwxr-xr-x   3 root root  4096 sep 17  2013 media
drwxr-xr-x   2 root root  4096 sep 22 12:33 mnt
drwxr-xr-x   2 root root  4096 feb 15  2011 opt
dr-xr-xr-x 108 root root     0 dic  3 13:09 proc
drwx------  33 root root  4096 dic  3 13:03 root
drwxr-xr-x  21 root root   880 dic 10 23:13 run
drwxr-xr-x   2 root root 12288 dic  3 13:08 sbin
drwxr-xr-x   2 root root  4096 dic  5  2009 selinux
drwxr-x--x   3 root root  4096 sep 17  2013 srv
drwxr-xr-x  13 root root     0 dic  3 14:09 sys
drwxrwxrwt  15 root root 12288 dic 11 00:17 tmp
drwxr-xr-x  11 root root  4096 nov  7  2013 usr
drwxr-xr-x  13 root root  4096 dic  3 13:08 var
lrwxrwxrwx   1 root root    29 dic  1 08:34 vmlinuz -> boot/vmlinuz-3.2.0-72-generic
```

*LECTOR: ¡Pues sí que contienen información los directorios!*

*Autor: Pues… te equivocas.*

En los sistemas tipo UNIX los directorios contienen múltiples entradas, sí, pero cada una de esas entradas lo único que almacena es una cadena (el nombre de la entrada) y un número que identifica una estructura de datos especial, que es la que almacena TODA la información relacionada con esa entrada, menos el nombre, claro. Esa estructura de datos recibe el nombre de nodo-i, en inglés **i-node**.

Para acceder a la mayor parte de la información contenida en una de estas estructuras podemos usar el programa ***stat***. Este programa usa una llamada al sistema que se llama **stat**.

No hay que marearse, una cosa es el programa y otra la llamada al sistema. Lo que puedo ejecutar desde el shell es el programa, que finalmente usará la llamada al sistema.

Para acceder al manual del programa: **man stat**

Para acceder al manual de la llamada al sistema: **man 2 stat**

En este caso recomiendo la lectura detallada de la llamada al sistema, que incluye la descripción pormenorizada de los campos de un nodo-i.

*Sí, vale, pero… ¿y esto qué tiene que ver con **ls –l**?*

La respuesta es sencilla, **ls** obtiene el nombre de la entrada del directorio, pero el resto de la información la obtiene haciendo **stat** sobre cada una de estas entradas, es decir, del nodo-i de las entradas. El programa ls solamente muestra un resumen de la información contenida en los nodos-i.

Veamos qué información de la que contiene un nodo-i me devuelve **stat**:

```
struct stat {
    dev_t         st_dev;     /* dispositivo en el que está este nodo-i */
    ino_t         st_ino;     /* número de inodo */
    mode_t        st_mode;    /* protección, tipo de fichero y otros*/
    nlink_t       st_nlink;   /* número de enlaces físicos */
    uid_t         st_uid;     /* UID del usuario propietario */
    gid_t         st_gid;     /* GID del grupo propietario */
    dev_t         st_rdev;    /* tipo dispositivo (si es nodo-i de dispositivo) */
    off_t         st_size;    /* tamaño total, en bytes */
    blksize_t     st_blksize;/* tamaño de bloque para sistema de ficheros de E/S */
    blkcnt_t      st_blocks; /* número de bloques asignados */
    time_t        st_atime;  /* hora último acceso */
    time_t        st_mtime;  /* hora última modificación */
    time_t        st_ctime;  /* hora último cambio */
};
```

Un nodo-i tiene un identificador único (**st_ino**) dentro de un dispositivo que pueda almacenar nodos-i (**st_dev**). Esto significa que si tengo dos entradas de directorio que tienen como identificador de nodo-i el mismo número, las entradas representan a dos ficheros que están en dos dispositivos diferentes o SON referencias al mismo fichero.

Una forma rápida de conocer el valor del nodo-i para las entradas de directorio sería hacer ls –i

El campo **st_nlink** indica cuantas entradas de directorio apuntan a este nodo-i. El valor más habitual es uno, aunque ya veremos que para algunos tipos de ficheros el valor siempre es mayor.

El campo **st_uid** indica el usuario responsable de este fichero, es el que se denomina propietario del fichero. Los procesos cuyo UID efectivo coincidan con este valor son los que menos restricciones tienen para el uso del fichero.

El campo **st_gid** indica el grupo del fichero. Todos los procesos que pertenezcan a este grupo, ya sea porque tienen su GID efectivo a este valor o porque este grupo es uno de sus grupos adicionales, tienen un conjunto propio de restricciones en el uso de este fichero.

Si un proceso tiene un UID efectivo que coincida con el propietario tendrá un conjunto de restricciones, sino lo tiene pero el grupo del fichero coincide con su GID efectivo o con alguno de sus GID adicionales, tendrá un segundo conjunto de restricciones. Finalmente habrá un tercer grupo de restricciones para aquellos procesos cuyo UID efectivo no coincida con el propietario del fichero, y los GID efectivos y adicionales no coincidan con el grupo propietario del fichero.

***PACIENCIA**:* Las restricciones las veremos en el siguiente punto de este documento

Los campos **st_blocks** y **st_size** nos indican el tamaño del fichero, el primero en bloques de **st_blksize** y el segundo en bytes. Hay que tener en cuenta que los archivos SIEMPRE ocupan en disco más que lo que dice el campo **st_size** debido a que hay que asignarles bloques enteros y en el mismo bloque no pueden haber datos de dos ficheros. Se podría comparar con un aparcamiento, hay N plazas, todas del mismo tamaño, sólo caben N coches normales, da igual que midan 2 metros o 5 metros (supongo plazas generosas pero no para camiones).

Los campos **st_atime**, **st_mtime** y **st_ctime** almacenan fechas sobre el uso de un archivo. El campo **st_atime** almacena la última vez que se accedio a los datos del fichero en lectura, **st_mtime** la última vez que se modificaron lo datos contenidos en el fichero (operación write) y **st_ctime** la última vez que hubo un cambio en la información que el sistema operativo guarda de un fichero (el nombre y el nodo-i) y siempre que esa modificación no sea exclusivamente en uno de estos tres campos (por ejemplo, si escribimos en el fichero pero no cambia el tamaño del fichero st_ctime permanece constante).

*Autor: Ya está explicado el nodo-i*

*Lector: No, no… te quedan dos campo: st_mode y st_rdev.*

*Autor: Bien, te has dado cuenta…Vamos a ello*

EL campo **st_mode** guarda la información que determina el tipo de fichero sobre el que estamos trabajando, no me refiero a si es un fichero de imagen **jpeg**, o un script de shell o MATLAB... me refiero a que tipo de fichero desde el punto de vista del sistema operativo. Además, guarda información adicional sobre qué acciones se pueden hacer sobre el fichero, quién puede hacerlas y si hay alguna cosa especial con el fichero. Y todo esto en 16 bits.

*Autor: Aquí están los permisos*

*Lector: POR FIN*

*Autor: Pero si acabamos de empezar…*

En este punto quiere dar nombre a los bits de este campo y dejamos su explicación detallada para cuando expliquemos permisos, porque este campo tiene tela...

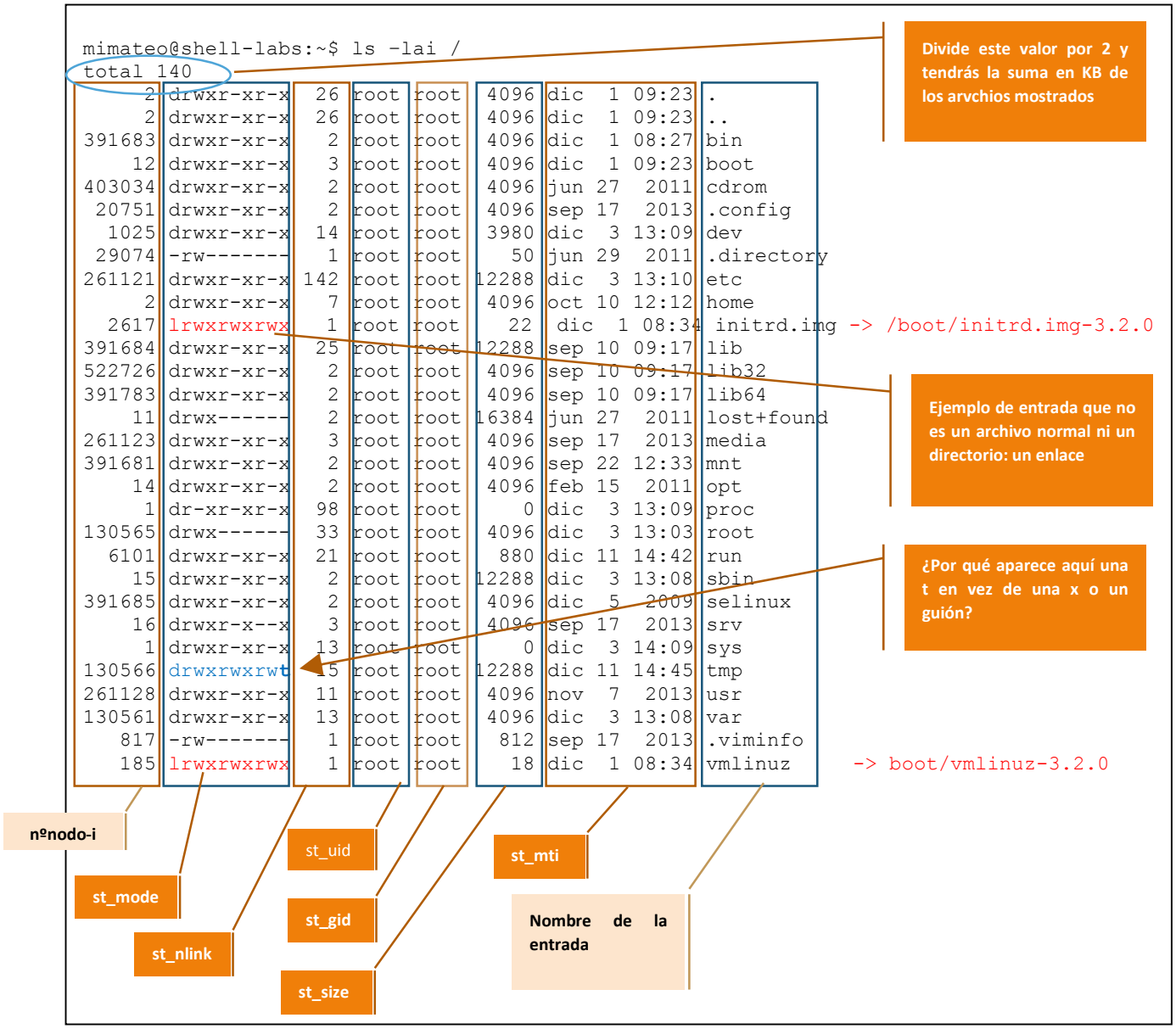
*Lector: Pues también te falta el campo **st_rdev***

*Autor: Eso sí que lo puedo explicar, pero primero déjame darle nombre a los bits de **st_mode***

| Bits | Descripción |
|---|---|
| 0..8 | Estos nueve bits definen las operaciones permitidas y quienes pueden hacerlas. Son tres grupos de tres bits.<br>Cada uno de los grupos se aplica a un conjunto diferente de usuarios.<br>Los bits en la misma posición en dos grupos diferentes tienen el mismo nombre y significado pero se aplican. Los nombres que reciben los bits:<br>• 0, 3 y 6 los llamaremos bits 'x'.<br>• 1, 4 y 7 los llamaremos bits 'w'<br>• 1, 4 y 7 los llamaremos bits 'r' |
| 9 | Siticky bit.<br>La utilización de este bit puede cambiar entre sistemas operativos porque NO es parte del estándar POSIX.<br>Se suele usar sólo en directorios, en los que significa que una entrada en ese directorio solamente puede ser modificada o borrada por el propietario del fichero al que apunta la entrada, por el propietario del directorio y por el superusuario.<br>Uso habitual: /tmp |
| 10 | SUID: bit de SetUID<br>Asociado a ficheros ejecutables, establece un mecanismo cambiar el UID Efectivo del proceso. |
| 11 | SGID: bit de SetGID<br>Asociado a ficheros ejecutables, establece un mecanismo cambiar el GID Efectivo del proceso<br>Asociado a directorios indican que todos los ficheros que se creen tendrán como GID de propietario el mismo valor que el directorio donde se encuentran.<br>Asociado a un fichero NO ejecutable indica que el acceso a este fichero se debe realizar |
| 12 al 15 | Codifican el tipo de fichero.<br>Los tipos de ficheros que Linux admite son:<br>• Fichero normal.<br>• Directorio.<br>• Enlace simbólico. Lo que en MS-Windows se llamaría acceso directo.<br>• Cola FIFO. Es el un tipo especial con el que, por ejemplo, se crean las tuberías<br>• Socket. Permite ver las conexiones de red como ficheros.<br>• Dispositivo orientado a carácter.<br>• Dispositivo orientado a bloque |

Y ahora sobre el campo **st_rdev**, este campo tiene sentido solamente cuando en el campo **st_mode** se ha especificado que este nodo-i en realidad apunta no apunta a un fichero sino a un dispositivo. Cuando esto ocurre el sistema necesita poder localizar ese dispositivo y su driver, esta información es la contenida en **st_rdev**. Este campo se divide internamente en dos números: el de mayor peso indica el tipo concreto de dispositivo y el menor identifica UN dispositivo concreto entre los de ese tipo. En otras palabras, con el de mayor peso el sistema operativo es capaz de encontrar un driver. Cuando el sistema operativo quiera usar ese dispositivo le hará peticiones al driver seleccionando pasándole como parámetro el número de

menor peso obtenido de **st_rdev**. Será responsabilidad del driver determinar cómo comunicarse con ese dispositivo concreto.




Y ahora podemos saber que significa cada campo mostrado por **ls –l**. Para entenderlo mejor volvemos a realizar el ls anterior pero añadiendo un par de opciones más para que también nos muestre los números de nodo-i y las entradas de directorio cuyos nombres empiezan por "**.**" (por convención de usuarios estas entradas se consideran archivos ocultos y muchos programas no las incluyen al mostrar el contenido de un directorio)

Si nos fijamos en la primera columna veremos varias entradas que tienen el mismo número de nodo-i. Eso significa que o esas entradas son enlaces físicos (hard links) al mismo fichero o apuntan a nodos-i en diferentes dispositivos. El dispositivo donde está un nodo-i lo podemos conseguir con el programa **stat**.

En la segunda columna se nos muestra una representación resumida del campo **st-mode**. Este resumen se basa en escribir una cadena que SIEMPRE tiene 10 caracteres que están organizados en 1 campo de un carácter y tres campos de tres caracteres. El primer campo, el de más a la izquierda, representa el tipo de fichero. Los posibles valores para este carácter son:

1. '**-**' : Esta entrada apunta a un fichero normal.
2. '**d**' : Esta entrada apunta a un directorio. El número de enlaces en un directorio siempre es un número mayor o igual que 2.
3. '**l**' : Esta entrada apunta a un enlace simbólico. El contenido de este fichero es la ruta al destino del enlace, el resto de bits del campo **st_mode** no tienen sentido y se suelen colocar a 1 los 9 de menor peso.
4. '**p**' : Esta entrada apunta a una cola FIFO, normalmente una tubería con nombre creada con la orden ***mkfifo*** o desde un proceso con la llamada *mkfifo*.
5. '**s**' : Esta entrada apunta a un socket.
6. '**c**' : Esta entrada apunta a un dispositivo orientado a carácter. Como en este caso el campo **st_rdev** tiene sentido y el campo **st_size** no, se muestre el primero separando el número mayor y menor con comas.
7. '**b**' : Esta entrada apunta a un dispositivo orientado a carácter. Como en este caso el campo **st_rdev** tiene sentido y el campo **st_size** no, se muestre el primero separando el número mayor y menor con comas.

Los 9 bits de menor peso del campo se muestran como 9 letras. Cuando uno de estos bits es un cero, aparece un guion en la posición correspondiente. Si el bit es un uno aparecerá una letra: 'x' para los bits 0, 3 y 6, 'w' para los bits 1, 4 y 7 y 'r' para los bits 2,5 y 8. Recordad que, aunque el significado de estas letras todavía no lo hemos explicado, si la entrada es un enlace simbólico no tienen efecto.

Los otros tres bits (sticky, SetUID y SetGID) no se muestran directamente. En vez de eso modifican la letra mostrada para los 9 bits de menor peso.

Si SetUID esta activada se modifica la 'x' de la posición 6 y se ponen una 's' en su lugar, si en la posición 6 hubiera un '-' se sustituye por 'S'. Con SetGID pasa algo parecido, pero el carácter que se modifica es el que está en la posición 3. Y lo mismo con el Sticky bit, pero en este caso sobre el carácter que está en la posición 0, sustituyendo 'x' por 't' y '-' por 'T'.

Para encontrar ejemplos de ficheros con SetUID y SetGID activados mirad los directorios /bin, /usr/bin.

El ejemplo más claro de scticky bit es el directorio /tmp

# 2 ACCIONES Y PERMISOS

Antes de empezar con los permisos debemos tener claro que elemento del sistema hace operaciones sobre los ficheros y procesos. Podríamos pensar que somos los usuarios quienes realizamos las acciones en el sistema, pero esto no sería cierto. Lo que hacemos los usuarios es generar eventos en un dispositivo de entrada que un proceso interpreta y realiza acciones en consonancia.

Al hablar de permisos muchas veces no se tiene esto en cuenta y se hablan de permisos de los usuarios. NO es así. Los que tienen permisos son los procesos, pero esos permisos se basarán en los UID y GID que el proceso tenga.

Lo que veremos en este apartado qué acciones puede intentar realizar un proceso sobre un fichero y qué hace el sistema para determinar si el proceso puede realizarla o no.

Hay que tener en cuenta que cada acción tendrá una serie de condicionantes para poder realizarse y no todos tendrán que ver con los permisos que pueda tener el proceso que intenta realizar la acción. Como ejemplo más claro cuando intentamos leer un fichero, si el fichero no existe será imposible leerlo.

## 2.1 ACCIONES

Las tres acciones básicas que se definen sobre los ficheros en Linux son TRES, ni una más. Etas tres acciones son:

- LEER: queremos acceder al contenido de un fichero.
- ESCRIBIR: queremos modificar el contenido de un fichero. Esto incluye agregar y quitar datos del fichero.
- EJECUTAR: si el archivo sobre el que se realiza no es un directorio, queremos que el sistema operativo intente ejecutarlo, si es un directorio queremos dar permisos de acceso adicionales. Aunque este bit se puede activar en cualquier archivo esto no significa que el sistema operativo pueda crear aplicaciones de ficheros que no las contienen.

Estas operaciones se comprueban en el instante en que un proceso empieza a realizarlas utilizando alguna llamada al sistema operativo. En el caso de LEER y ESCRIBIR se suele hacer como consecuencia de un **open()** y en el caso de EJECUTAR al realizar algún tipo de **exec()**. Hay otras llamadas al sistema que también necesitarán comprobar que pueden realizar alguna operación, pero siempre basadas en estas tres.

### 2.1.1 Acciones sobre directorios

Mucha gente piensa que las operaciones como copiar, mover o borrar un archivo tienen que ver únicamente con el archivo, y tampoco piensan que para llegar a un directorio hay que recorrer una ruta.

La realidad es que para hacer todas estas acciones hay que pasar por unos ficheros intermedios de un tipo especial: LOS DIRECTORIOS.

Estos ficheros soportan las mismas acciones básicas que cualquier otro fichero. Obviamente NO se pueden ejecutar, por lo que los permisos relacionados con ejecutar tienen un significado diferente en el caso de los directorios.

Las acciones que puedo hacer sobre los directorios son las siguientes:

- LEER: queremos acceder al contenido del directorio, es decir, los nombres de las entradas y los números de nodos-i.
- UTILIZAR COMO PARTE DE UNA RUTA: este acción incluye el cambiar a ese directorio, es decir, que un proceso pueda utilizar ese directorio como directorio de trabajo actual.
- ACCEDER A LOS NODOS-I: tener acceso a los nodos-i de los ficheros. Etsa acción será necesaria para comprobar las acciones que podemos realizar sobre los fichero, ejecutar un **stat** sobre el

fichero o realizar operaciones que impliquen un cambio en esos nodos-i (si no los puedo leer no podré modificarlos)

- ESCRIBIR: queremos modificar el contenido directorio. Esto incluye agregar y quitar entradas del directorio.

Son cuatro acciones, para simplificar el posterior tratamiento de permisos las dos acciones intermedias compartirán SIEMPRE los permisos. Es decir, si puedo hacer un cd sobre un directorio podré ejecutar una llamada stat sobre cualquier entrada de ese directorio.

## 2.2 PERMISOS

En UNIX tendremos tres tipos de permisos, uno por cada una de las acciones que hemos definido para acceder a los ficheros: LECTURA (r), ESCRITURA (w) y EJECUCIÓN(x).

*Lector: ¿rwx? no eran esa las letras que se mostraban en el **ls -l***

*Autor: Exacto… muy bien…. ;-)*

*Lector: Pero es que habían nueve letras, y las x se podían convertir en t o en S*

*Autor: A eso voy, a ver si lo hago bien.*

> **¿Te hace falta seguir aquí?**
>
> Llegados a este punto creo que cualquiera sería capaz de entender el manual cuando explica lo que son los permisos, los propietarios o las modificaciones impuestas por los bits SetUID, SetGID o el Sticky. Por ejemplo en las siguientes direcciones:
>
> http://www.comptechdoc.org/os/linux/usersguide/linux_ugfilesp.html
>
> http://www.ee.surrey.ac.uk/Teaching/Unix/unix5.html

Recordemos que habían nueve bits en el campo **st_mode** de un nodo-i pendientes de explicar. No es casualidad que en su momento habláramos de tres grupos de tres bits. Los tres bits se corresponden con las tres posibles acciones que podemos hacer sobre cualquier fichero.

*Lector: Y los tres grupos*

El grupo más a la derecha (mayor peso) se refiere a las acciones que el propietario del fichero puede hacer en el fichero: leer los datos, escribir/modificar los datos o intentar ejecutar. Un uno en un bit indica que el propietario puede realizar la acción.

*Lector:¿No habías dicho que los usuarios no tienen permisos?*

Cuando se habla de propietario de fichero en este contexto, se está hablando de un proceso cuyo UIDEfectivo coincida con el campo **st_uid** del nodo-i del fichero.

El siguiente grupo (bits 3,4 y 5) se refiere a las acciones permitidas a los procesos en los que podamos encontrar el **st_gid** del fichero en el GID efectivo o en el conjunto de grupos suplementarios.

A los otros procesos se le permiten las acciones definidas en el último grupo de tres bits (bits 0, 1 y 2).

Ejemplo 1:

```
-rw-r-----   1 root dialing    50 jun 29  2011 mod_perm
```

- Están activados los bits 5, 7 y 8.
- Tiene un tamaño de 50 bytes.
- La última vez que se modificaron los datos contenidos fue el 29 de junio de 2011.
- El st_uid es root ➔ A los procesos con UID Efectivo igual a *root* se les permite las acciones de lectura y modificación de los datos del fichero
- El st_gid es *dialing* ➔ A los procesos con GID Efectivo igual a *dialing* o aquellos que entre los grupos adicionales tengan ese valor, se les permite la acción de lectura de los datos del fichero
- Los procesos que no cumplan ninguna de las dos condiciones anteriores no pueden realizar acciones sobre el fichero.

Ejemplo 2:

```
-rwxr-x--x   1 juan advanced  4590 jun 29  2011 mi_servicio
```

- Están activados los bits 0, 3, 5, 6, 7 y 8.
- Tiene un tamaño de 4590 bytes.
- La última vez que se modificaron los datos contenidos fue el 29 de junio de 2011.
- El st_uid es juan ➔ A los procesos con UID Efectivo igual a *juan* se les permite las acciones de lectua, modificación y ejecución de los datos del fichero
- El st_gid es advanced ➔ A los procesos con GID Efectivo igual a *advanced* o aquellos que entre los grupos adicionales tengan ese valor, se les permite las acciones de lectura de los datos del fichero y la ejecución del fichero.
- Los procesos que no cumplan ninguna de las dos condiciones anteriores pueden realizar la acción de ejecución. No pueden leer o modificar los datos, pero el fichero lo pueden escribir.

Ejemplo 3:

```
----------   1 juan juan    5550 jun 29  2010 secreto
```

- Están desactivados todos los bits.
- Tiene un tamaño de 5550 bytes.
- La última vez que se modificaron los datos contenidos fue el 29 de junio de 2010.
- El st_uid es juan
- El st_gid es juan
- Ningún proceso puede realizar ninguna de las acciones definidas sobre el proceso.

## 2.3 PERMISOS SOBRE DIRECTORIOS

Como para los ficheros, los directorios tendrán tres tipos de permisos, lo que significa que uno de estos permisos tendrá que usarse para permitir dos de las cuatro acciones posibles sobre los directorios. Estos permisos están agrupados según el UIDEfectivo y los GID del proceso que intente hacer la acción, lo que permite tener tres conjuntos de permisos, tres conjuntos de r,w y x. Así estos permisos:

- Permiso ‘r’: permite leer los datos contenidos en el directorio, es decir, los nombres de las entradas y los números de nodo-i de cada una de ellas.
- Permiso ‘x’: permite utilizar este directorio como directorio de trabajo, permite utilizar este directorio como parte de una ruta más larga, permite el acceso a los nodos-i de las entradas. Sin este permiso no podremos hacer un cd o un ls –l sobre el directorio. Este permiso sólo tiene sentido se tenemos permiso ‘r’ en el directorio.
- Permiso ‘w’: solamente tiene sentido si tenemos permiso ‘x’ en el directorio. Este permiso permite realizar operaciones de modificación de las entradas del directorio: cambiar el nombre, añadir nuevas y eliminar las existentes. Sin este permiso no podremos realizar lo que estamos acostumbrados a llamar “*borrar un fichero”.*

### 2.3.1 Ejemplos de órdenes y permisos en directorios

**Orden ls /tmp/entrada1**

Para ejecutar esta orden necesitamos permisos de lectura y ejecución en los directorio **/** y **/tmp**. En ambos casos es porque forman parte de una ruta y necesitamos conocer el nodo-i del siguiente componente de la ruta para saber cuándo parar.

Si **entrada1** es un fichero, no necesitamos nada más. Si es un directorio necesitamos permisos de lectura para poder mostrar los nombres e las entradas de directorio contenidas en el mismo.

**Orden cd /tmp/entrada1**

Para ejecutar esta orden necesitamos permisos de lectura y ejecución en los directorio **/** y **/tmp**. En ambos casos es porque forman parte de una ruta y necesitamos conocer el nodo-i del siguiente componente de la ruta para saber cuándo parar.

Si **entrada1** no es un directorio la orden falla. Si es un directorio necesitamos permisos de lectura y ejecución para hacerlo directorio actual

**Orden rm /tmp/entrada1**

Esta orden elimina la entrada con nombre **entrada1** del directorio /tmp. Supondremos que la entrada existe.

Necesitamos permisos de lectura y ejecución en los directorio **/** y **/tmp**. En ambos casos es porque forman parte de una ruta y necesitamos conocer el nodo-i del siguiente comonente de la ruta para saber cuándo parar.

Cuando accedemos al nodo-i de **entrada1** se comprueba que NO sea un directorio, ya que el programa **rm** por defecto no puede borrar directorios, hay opciones especiales para hacerlo.

Para poder borrar esta entrada SOLAMENTE necesitamos que en **/tmp** también tengamos permisos de escritura. Eliminar una entrada significa escribir en el directorio, no accedemos a los datos del fichero.

Como efecto colateral se decrementa el número de enlaces en el nodo-i. Si este número llega a 0 el sistema operativo decide que el nodo-i ya no es accesible y lo pone como libre, liberando al mismo tiempo TODOS los recursos que tenía hasta ese momento.

Esto es lo más cercano a borrar un fichero que tenemos en UNIX.

## 2.4 OTRAS ACCIONES

Existen otra acción posible de la que no hemos hablado: cambiar datos del nodo-i directamente. Esta acción se define como la capacidad de cambiar los campos st_mode, st_uid y st_gid de un fichero. Los demás campos del nodo-i están definidos por el sistema operativo y se cambian automáticamente al realizar otras acciones en el sistema.

Para cambiar el campo st_mode se utiliza la llamada al sistema **chmod()** o un programa con el mismo nombre, **chmod**. Para que un proceso pueda modificar el campo st_mode de un fichero debe cumplir la condición de que su UIDEfectivo sea el mismo que el valor contenido en **st_uid**. Hay que destacar que en este caso NO es necesario tener permisos para otra operación, dicho de otro modo, en el ejemplo 3 del punto 2.3 podemos cambiar el campo st_mode aunque los bits inicialmente están todos a cero.

El intento de cambiar los campos st_uid y st_gid es diferente. Para esta acción se pueden utilizar la llamada chown() o los programas chown o chgroup. Para poder usarlas, un proceso tiene que cumplir la misma restricción que para cambiar el campo st_mode, es decir, su UIDEfectivo debe coincidir con el campo st_uid.

Esta acción es peligrosa, ya que si no hubiera más restricciones podríamos asignar la propiedad y responsabilidad de un fichero a cualquier usuario de la máquina. Por ese motivo un proceso que no pertenezca al superusuario sólo puede cambiar el campo st_gid a los valores contenidos en los campos GID del proceso (efectivo, real, salvado, fs o adicionales). Además esta acción tiene efectos colaterales ya que implica poner a cero los bits SUID y SGID del fichero.

### 2.4.1 chmod

Existen diferentes formas de llamar a chmod, básicamente agrupadas en dos tipos: pasando los modos en forma numérica y pasándolos en forma simbólica.

En el primer caso se pasan los 9 bits que queremos que tenga como “permisos” el fichero tras la llamada. Estos 9 bits se codifican como 3 dígitos octales, es decir, se pasa un número octal de tres cifras, cada una de las cuales representa 3 bits.

En el segundo se utilizan las letras u,g y o para indicar el grupo que se quiere cambiar (user, group y others) y las letras r,w,x para indicar los bits que se quieren modificar. Para poner un permiso a cero pondremos un guion (-) delante de la letra que lo identifica. Para activarlo pondremos un más (+) o nada.

Ejemplo 1:

```
chmod 777 pepe
```

Cambiamos los permisos de pepe poniendo los nueve bits a 1, lo que significa que en un ls veremos como permisos rwxrwxrwx.

Ejemplo 2:

```
chmod 600 juan
```

Cambiamos los permisos de pepe poniendo los dos 2 bits de mayor peso a 1 y el resto a 0, lo que significa que en un ls veremos como permisos rw-------.

Ejemplo 3:

```
----------   1 juan juan    5550 jun 29  2010 secreto
chmod u+r+w+x,g+r+x,o+x secreto
```

Cambiamos los permisos de secretos poniendo los bits 1, 4,6, 7, 8 y 9 a 1, lo que significa que en un ls veremos como permisos rwxr-x--x.

Si luego hacemos

```
chmod g-x,o-x secreto
```

Dejaremos los permisos como rwxr-----

# 3 OTROS MODIFICADORES

Hay tres bits en el campo st_mode de los que hemos hablado pero no hemos dicho qué utilidad tienen en el sistema. Esto no significa que no sean importantes, al contrario, pero hemos preferido dejarlos para el final para que sea más sencillo explicar cómo se utilizan.

## 3.1 SETUID Y SETGID

La utilidad básica de estos dos bits es conseguir que se ejecuten programas con unos los permisos que tendría un determinado usuario (un proceso con UID igual al de ese usuario) independientemente del UID del proceso que ejecuta dicho programa.

Un ejemplo de esta situación es el programa **mount**, programa que permite a un usuario conectar y desconectar discos al sistema. En principio esta acción solamente está autorizada a realizarla el superusuario (UID==0 e identificador **root**), pero es muy útil que en determinadas situaciones algunos usuarios sean capaces de realizarla sin que sea necesario proporcionarles todos los permisos que tiene el usuario **root**.

El significado de estos bits en un programa ejecutable sería:

- Si el bit SETUID está activado en un fichero que un proceso intenta ejecutar, si el proceso tiene los permisos para realizar la ejecución, el proceso cambia su UID efectivo al UID del propietario del fichero.
- Si el bit SETGID está activado en un fichero que un proceso intenta ejecutar, si el proceso tiene los permisos para realizar la ejecución, el proceso cambia su GID efectivo al GID del grupo propietario del fichero.

Como el UID y el GID reales no cambian, el programa en ejecución puede comprobar su valor para ajustar su operación a las acciones inicialmente permitidas al proceso original. Volviendo al ejemplo de **mount**, si el proceso tiene **root** como UID real no se hacen más comprobaciones y se intenta realizar la acción pedida, en cambio, si el UID real no era ese se utilizan los filtros definidos en /etc/fstab para determinar si la operación pedida por el usuario se puede intentar llevar a cabo o no.

Aunque puede parecer que estos bits sirven para dar más permisos, en algunos casos la idea es la opuesta, restringir los permisos de un proceso al hacer determinadas operaciones.

En cualquier caso, construir un programa que saque partido de estos bits es difícil porque puede generar problemas de seguridad en el sistema, sobre todo si el programa tiene como UID propietario **root** (recordemos que los programas son ficheros ejecutables).

### 3.1.1 SGID en directorios

Los directorios no pueden ser programas, por lo que en principio no tiene sentido activar los bits SetUID o SetGID en ellos.

Sin embargo, el SetGID sí que se utiliza en algunas versiones de UNIX, entre ellas Linux.

Si el bit SGID está activo en un directorio, cualquier fichero que se cree en ese directorio heredará el grupo propietario del directorio, es decir, todos los ficheros que se creen en él no pondrán en su campo GID el identificador UIDEfectivo del proceso que ls crea, sino el GID del directorio en el que se crean.

OJO: este comportamiento depende de la versión de UNIX en la que estemos.

## 3.2 SITCKY BIT

La utilización de este bit del campo **st_mode** no está definida en el estándar POSIX, lo que significa que su existencia o utilidad puede variar de un sistema operativo a otro.

Sin embargo, en muchos UNIX, entre ellos LINUX, su uso principal es limitar la posibilidad de borrar entradas de directorio. Recordemos que para eliminar una entrada de directorio sólo se consulta el nodo-i de la entrada para comprobar si la entrada es un directorio o un fichero (los directorios no se pueden borrar con las mismas órdenes). Esto significa que en un directorio que tengamos permisos de escritura podemos eliminar cualquier fichero.

Esto nos introduce en el problema de /tmp, un directorio que está pensado para que los procesos creen y destruyan información temporal. Para conseguirlo este directorio tiene como máscara de permisos RWX para el propietario, para el grupo y para otros… TODOS LOS PROCESOS PUEDEN ESCRIBIR.

> *Lector: No veo ningún problema*
>
> *Autor: Si quieres que la información temporal que has guardado la pueda eliminar otro usuario, no existe el problema.*
>
> *Lector: Eso no se puede permitir, que haya un /tmp para cada usuario, o dame otra solución*
>
> *Autor: No hacen falta otros* tmp… para eso está el bit que paso a explicarte*:*

Si un directorio tiene activado el sticky bit la acción “Eliminar entrada” tiene un nuevo requisito: el proceso que elimina tiene que tener un UID Efectivo igual al UID de propietario del fichero. Esto significa que un fichero en /tmp solamente lo podrá eliminar un proceso del usuario que creo dicho fichero.

*Autor: Y eso es todo… o el principio solamente. Depende de cómo lo mires*

# 4 PÁGINAS WEB RELACIONADAS

- http://content.hccfl.edu/pollock/AUnix1/FilePermissions.htm
- http://www.ee.surrey.ac.uk/Teaching/Unix/unix5.html
- http://www.linuxnix.com/2011/12/sgid-set-sgid-linuxunix.html
- http://www.codecoffee.com/tipsforlinux/articles/028.html
- http://man7.org/
- http://man7.org/linux/man-pages/index.html

Y por qué no, las nuestras:

- http://armpower.blogs.upv.es/
- http://acomp.disca.upv.es/acso1/