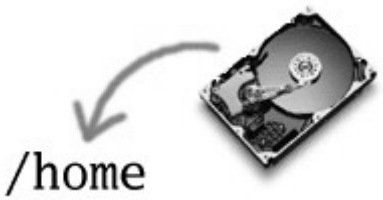
Lección 20

MOUNT / UMOUNT



Indice

Introducción

Un sistema de ficheros esta compuesto por las funciones y las estructuras de datos que emplea el sistema operativo (en nuestro caso, Linux) para organizar los ficheros en disco. El Sistema de Ficheros se ubica en una partición o en un disco. Linux soporta varios tipos de sistemas de ficheros. Entre los más importantes podemos destacar los siguientes:

- MINIX: el más antiguo, presume de ser el más seguro, pero es bastante limitado en las características que proporciona. Un sistema de ficheros de este tipo solo puede tener 64 MB.
- EXT2: es el sistema de ficheros nativo de Linux. Está diseñado para ser compatible con versiones anteriores, así que las nuevas versiones del código del sistema de ficheros no requerirán rehacer los sistemas de ficheros existentes.
- EXT3: es una modificación del ext2 para añadirle funcionalidades de journaling.
- VFAT: este tipo permite utilizar sistemas de ficheros de Windows (FAT, FAT32), y actualmente está soportado el sistema de ficheros de Windows NT, pero sólo fiable en sólo-lectura.
- Iso9660: es el sistema de ficheros estándar para CD-ROM.
- NFS: un sistema de ficheros en red que permite compartir sistemas de ficheros entre diferentes máquinas conectadas en red y tratarlos de forma local.

Existe también un sistema de ficheros especial denominado proc, y que es accesible vía el directorio /proc, el cual no es realmente un sistema de ficheros. El sistema de ficheros /proc permite acceder fácilmente a ciertas estructuras de datos del núcleo, como es la lista de procesos. Convierte estas estructuras de datos en algo parecido a un sistema de ficheros y por tanto da la posibilidad de manipularlas con las herramientas habituales de manipulación de ficheros. Hay que tener en cuenta que aunque se le denomine sistema de ficheros, ninguna parte del sistema de ficheros /proc toca el disco. Existe únicamente en la memoria principal.

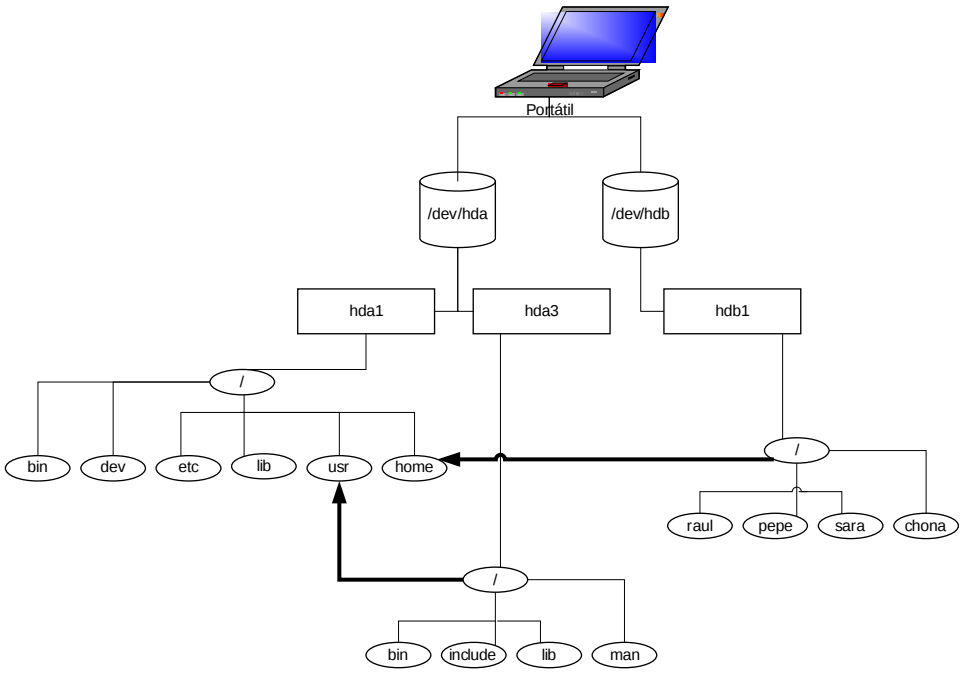
Un concepto clave del diseño de software de E/S es la independencia del dispositivo, es decir, emplear las mismas instrucciones para leer un fichero sin importar si éste reside en el disco duro o en el CDROM. De esta manera queda la responsabilidad del tratamiento específico de cada dispositivo al sistema operativo, al igual que los posibles errores derivados de este subnivel.
Los archivos forman un árbol jerárquico compuesto por nodos (los directorios) y hojas (los demás tipos de archivos). Esta visión de árbol es una visión lógica que no tiene en cuenta la organización física de los datos en los discos. Aunque Linux presenta al usuario la visión de un solo árbol de archivos, es frecuente que esta jerarquía esté compuesta por varios sistemas de archivos situados sobre particiones o discos diferentes.

El ensamblado lógico de los diferentes sistemas de archivos se realiza por una operación de montaje. Ésta se efectúa generalmente al arrancar el sistema, donde los sistemas de fichero listados en */etc/fstab* son montados automáticamente (a no ser que se indique que requieren un montado manual), pero también puede lanzarse durante el funcionamiento de Linux, por ejemplo, para acceder a datos presentes en soportes removibles, como los CD-ROM, USB, etc.

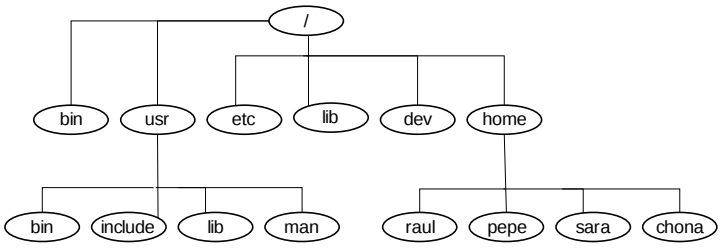
La figura representa una operación de montaje efectuada en la inicialización del sistema. En esta figura existen dos sistemas de archivos: el sistema de archivos raíz y el sistema de archivos que contiene la jerarquía /usr. Cuando se efectúa el montaje, por el comando **mount** que utiliza la llamada *mount*, el contenido del segundo sistema de archivos se vincula al sistema de archivos raíz y es accesible bajo el directorio /usr.

Mount

Antes de poder utilizar un sistema de ficheros, este debe de ser montado. A diferencia de otros sistemas operativos y como cualquier UNIX, Linux emplea una jerarquía (árbol) de directorios único. Por tanto la operación de montaje hará que los contenidos de un sistema de ficheros nuevo parezcan los contenidos de un subdirectorio existente de algún sistema de ficheros ya montado.
Imaginemos dos sistemas de ficheros separados, cada uno con su propio directorio raíz. Cuando el segundo sistema de ficheros se monte bajo el directorio /home en el primer sistema de ficheros, se obtendrá un único árbol de directorios.



El comando MOUNT realiza la fusión de dos sistemas de ficheros, así, una vez que el sistema de ficheros está montado forma parte del árbol de directorios. Es decir, el MOUNT permite que sistemas de archivos independientes sobre dispositivos diferentes se «peguen» para formar un solo árbol del sistema.



Para montar un sistema de ficheros se debe especificar el dispositivo a montar y el punto de montaje dentro del árbol de directorios:

mount [-t <tipo>] <dispositivo> <punto_de_montaje> [-o <opciones>]

```
mount –t vfat /dev/fd0 /mnt/floppy
mount –t iso9660 /dev/hdb0 /mnt/cdrom
mount /dev/hda1 /mnt/discoduro
mount /dev/cdrom /mnt/cdrom
```

La opción -t le indica al comando mount el tipo de sistema de ficheros que se va a montar, el cual deberá estar soportado por el núcleo. Los otros dos argumentos son el fichero dispositivo correspondiente a la partición que contiene el sistema de ficheros, y el directorio sobre el que será montado. El directorio sobre el que se monta no debe estar vacío, pero sí debe existir. Cualquier fichero que haya en ese directorio será inaccesible mientras el sistema de ficheros esté montado.

Si por ejemplo no deseáramos que nadie pudiera escribir en el sistema de ficheros se podría haber usado la opción -r de mount para indicar que es un sistema de ficheros de sólo lectura. Esto obligará al núcleo a detener cualquier intento de escritura sobre el sistema de ficheros y también detendrá cualquier actualización de los tiempos de acceso de los inodos.

En este momento alguien puede preguntarse como es posible que el sistema de ficheros raíz se monte, ya que no puede ser montado en otro sistema de ficheros. Entonces si el sistema de ficheros raíz no puede ser montado, el sistema no arrancará nunca. La respuesta está en que ese sistema de ficheros que se monta como root está compilado en núcleo o definido utilizando LILO o el comando rdev.

mount --bind/--move
Se puede montar un subárbol del árbol de directorios de nuestro sistema en múltiples lugares. Se puede hacer uso de una característica de los nuevos núcleos 2.6. Podemos usar:

mount --bind /usr/local/bin /export/bin

Esta opción de mount permite montar un subárbol de nuestro sistema de ficheros en otro punto del mismo. El directorio /export/bin debe existir (como si se tratara de otro punto de montaje cualquiera), pero una vez ejecutado el comando si accedemos al directorio veremos el contenido de /usr/local/bin. Desde luego esto tiene su utilidad, y una de ellas es la de poder tener un directorio /export con los subdirectorios de nuestro sistema de ficheros que queramos exportar. Otra opción de similares características es:

mount --move /export/bin /export/bin2

Que en este caso mueve el punto de montaje creado anteriormente con mount --bind a otro sitio.

Cuando un sistema de ficheros no se necesita, se puede desmontar utilizando la orden umount, la cual toma como argumento el fichero dispositivo que define la partición que alberga al sistema de ficheros o el nombre del directorio sobre el que se ha montado.

Un ejemplo claro son los disquetes, que no deberían ser extraídos sin previamente haber desmontado el sistema de ficheros. Ya que debido al caché de disco los datos no son necesariamente escritos hasta que se desmonta.

Los dispositivos se encuentran dentro de /dev. Así es como se designan los más comunes:

- fd0 Primera unidad de disquetes (a: en sistemas MS-DOS y Windows).
- fd1 Segunda unidad de disquetes (b: en sistemas MS-DOS y Windows).
- hda Primer disco duro IDE (Primary Master).
- hda0 Primera partición del primer disco duro IDE (Primary Master).
- hda1 Segunda partición del primer disco duro IDE (Primary Slave).
- hdb0 Primera partición del segundo disco duro IDE (Secondary Master).
- hdb1 Segunda partición del segundo disco duro IDE (Secondary Slave).
- sda Primer disco duro SCSI.
- sda1 Primera partición del primer disco duro SCSI.
- sdb4 Cuarta partición del segundo disco duro SCSI.
- scd0 Primera unidad de CD-ROM SCSI.
- scd1 Segunda unidad de CD-ROM SCSI.
- sga Primer dispositivo genérico SCSI (scanner, etc.).

- sgb Primer dispositivo genérico SCSI.
- sg0 Primer dispositivo genérico SCSI en sistemas nuevos.
- sg1 Segundo dispositivo genérico SCSI en sistemas nuevos.

La llamada al sistema MOUNT, monta el sistema de archivos presente en el dispositivo cuyo nombre se pasa en el parámetro ***specialfile***. El parámetro ***dir*** indica el nombre del punto de montaje, es decir, el nombre del directorio a a partir del cual el sistema de archivos es accesible. El tipo del sistema de archivos se pasa en el parámetro ***filesystemtype***, y se trata de una cadena de caracteres que representa un tipo de sistema de archivos conocido por el núcleo Linux, como “minix”, “ext2”, “iso9660”... Los parámetros ***rwflag*** y ***data*** especifican las opciones de montaje y sólo se tienen en cuenta si los 16 bits de mayor peso de ***rwflag*** son iguales al valor 0xC0ED.

```
int umount (const char *specialfile);
int umount (const char *dir);
```

Llamadas al sistema:

```
 280  asmlinkage long sys_mount(char __user *dev_name, char __user *dir_name,
 281                            char __user *type, unsigned long flags,
 282                            void __user *data);
 283  asmlinkage long sys_umount(char __user *name, int flags);
```

linux+v2.6.29/include/linux/syscalls.h

Algunas de las opciones de montaje son:

```
 117/*
 118 * These are the fs-independent mount-flags: up to 32 flags are supported
 119 */
 120#define MS_RDONLY        1       /* Mount read-only */
 121#define MS_NOSUID        2       /* Ignore suid and sgid bits */
 122#define MS_NODEV         4       /* Disallow access to device special files */
 123#define MS_NOEXEC        8       /* Disallow program execution */
 124#define MS_SYNCHRONOUS   16      /* Writes are synced at once */
 125#define MS_REMOUNT       32      /* Alter flags of a mounted FS */
 126#define MS_MANDLOCK      64      /* Allow mandatory locks on an FS */
 127#define MS_DIRSYNC       128     /* Directory modifications are synchronous */
 128#define MS_NOATIME       1024    /* Do not update access times. */
 129#define MS_NODIRATIME    2048    /* Do not update directory access times */
 130#define MS_BIND          4096
 131#define MS_MOVE          8192
 132#define MS_REC           16384
 133#define MS_VERBOSE       32768   /* War is peace. Verbosity is silence.
 134                                    MS_VERBOSE is deprecated. */
 135#define MS_SILENT        32768
 136#define MS_POSIXACL      (1<<16) /* VFS does not apply the umask */
 137#define MS_UNBINDABLE    (1<<17) /* change to unbindable */
 138#define MS_PRIVATE       (1<<18) /* change to private */
 139#define MS_SLAVE         (1<<19) /* change to slave */
 140#define MS_SHARED        (1<<20) /* change to shared */
 141#define MS_RELATIME      (1<<21) /* Update atime relative to mtime/ctime. */
 142#define MS_KERNMOUNT     (1<<22) /* this is a kern_mount call */
 143#define MS_I_VERSION     (1<<23) /* Update inode I_version field */
 144#define MS_ACTIVE        (1<<30)
 145#define MS_NOUSER        (1<<31)
```

linux+v2.6.29/include/linux/fs.h

Es posible combinar estas diferentes opciones mediante una operación O binaria (operador | del lenguaje C). Una constante particular puede utilizarse para modificar las opciones de montaje de un sistema de archivos montado anteriormente.
El parámetro data apunta a una cadena de caracteres que contiene opciones suplementarias. El contenido de esta cadena es dependiente del tipo de sistema de archivos.
La llamada umount desmonta un sistema de archivos montado anteriormente por una llamada a mount. Acepta como parámetro tanto un nombre de archivo especial (parámetro ***specialfile***) como un nombre de punto de montaje (parámetro ***dir***).

Estructuras de datos

VFSMOUNT

Antes de poder explicar el MOUNT o el UMOUNT, debemos conocer la estructura de datos en la que estas funciones se fundamentan. Los sistemas de ficheros montados se describen por una estructura llamada VFSMOUNT, que se encuentra definida en /linux/mount.h

```
 38struct vfsmount {
 39      struct list_head mnt_hash;
 40      struct vfsmount *mnt_parent;            /* fs we are mounted on */
 41      struct dentry *mnt_mountpoint;          /* dentry of mountpoint */
 42      struct dentry *mnt_root;                /* root of the mounted tree */
 43      struct super_block *mnt_sb;             /* pointer to superblock */
 44      struct list_head mnt_mounts;            /* list of children, anchored here */
 45      struct list_head mnt_child;             /* and going through their mnt_child */
 46      int mnt_flags;
 47                                              /* 4 bytes hole on 64bits arches */
 48      const char *mnt_devname;                /* Name of device e.g. /dev/dsk/hda1 */
 49      struct list_head mnt_list;
 50      struct list_head mnt_expire;            /* link in fs-specific expiry list */
 51      struct list_head mnt_share;             /* circular list of shared mounts */
 52      struct list_head mnt_slave_list;        /* list of slave mounts */
 53      struct list_head mnt_slave;             /* slave list entry */
 54      struct vfsmount *mnt_master;            /* slave is on master->mnt_slave_list */
 55      struct mnt_namespace *mnt_ns;           /* containing namespace */
 56      int mnt_id;                             /* mount identifier */
 57      int mnt_group_id;                       /* peer group identifier */
 58      /*
 59       * We put mnt_count & mnt_expiry_mark at the end of struct vfsmount
 60       * to let these frequently modified fields in a separate cache line
 61       * (so that reads of mnt_flags wont ping-pong on SMP machines)
 62       */
 63      atomic_t mnt_count;
 64      int mnt_expiry_mark;                    /* true if marked for expiry */
 65      int mnt_pinned;
 66      int mnt_ghosts;
 67      /*
 68       * This value is not stable unless all of the mnt_writers[] spinlocks
 69       * are held, and all mnt_writer[]s on this mount have 0 as their ->count
 70       */
 71      atomic_t __mnt_writers;
 72};
```

linux+v2.6.29/include/linux/mount.h

Y la descripción de alguno de sus campos significan lo siguiente:

mnt_parent	Sistema de ficheros que va a ser mountado	mnt_mounts	Lista de hijos anclados en ese punto.
mnt_mountpoint	Dirección del punto de montaje	mnt_child	Examinar los hijos montados.
mnt_root	Raíz del arbol del sistema de ficheros	mnt_flags	Opciones
mnt_sb	Puntero Super bloque	mnt_devname	Nombre del dispositivo a montar.
mnt_list		mnt_expire	Enlace a la lista de dispositivos expirados. Fs-specific.
mnt_share	Lista circular de dispositivos compartidos y montados.	mnt_slave_list	Lista de dispositivos esclavos montados.
mnt_master	Maestro de la lista mnt_slave_list	mnt_ns	Espacio de nombres.
mnt_count		mnt_expiry_mark	Marca de dispositivo expirado.

El campo mnt_sb apunta a la estructura superbloque que controla el sistema de ficheros. Contiene la información siguiente:

- Tamaño del bloque de datos
- Punteros a estructuras de manejadores
- Punteros al inodo raíz de dicho sistema de ficheros
- Punteros al registro del sistema de ficherosInformación específica dependiente del

sistema de ficheros: lista de bloques libres, modos de acceso, etc.

Funciones

La función que implementa la llamada al sistema MOUNT realiza básicamente cuatro acciones diferenciadas:

- Copiar las opciones de montaje desde el espacio de direccionamiento del proceso actual al espacio de direccionamiento del núcleo, con la función COPY_MOUNT_OPTIONS.
- Se copia la dirección del punto de montaje del espacio de usuario al espacio del núcleo por medio de la función GETNAME, la cual realiza el chequeo de errores.
- Llama a la función DO_MOUNT que es realmente la encargada de realizar el montaje. Más adelante veremos que esta afirmación no es del todo cierta, ya que la función más importante en este caso, por ser la que realmente monta el sistema de ficheros, es VFS_KERN_MOUNT, si bien, si es cierto que es DO_MOUNT la que desencadena todo el proceso de montaje.
- Y por último, libera la memoria usada para las opciones.

La llamada al sistema se encuentra implementada en el fichero /fs/namespace.c por medio de una macro definida en include/linux/syscalls.h SYSCALL_DEFINE5 y “mount” como parámetro.

```
2050SYSCALL_DEFINE5(mount, char __user *, dev_name, char __user *, dir_name,
2051            char __user *, type, unsigned long, flags, void __user *, data)
2052{
2053     int retval;
2054     unsigned long data_page;
2055     unsigned long type_page;
2056     unsigned long dev_page;
2057     char *dir_page;
2058
```

```
2059     retval = copy_mount_options(type, &type_page);
2060     if (retval < 0)
2061          return retval;
```

Pasa al espacio de direccionamiento del núcleo el tipo de dispositivo a montar. Si se produjera algún error, retorna de la función, devolviendo el error indicado por la función copy_mount_options.

```
2063     dir_page = getname(dir_name);
2064     retval = PTR_ERR(dir_page);
2065     if (IS_ERR(dir_page))
2066          goto out1;
```

Pasa al espacio de direccionamiento del núcleo la dirección del punto de montaje. Si se produce un error, salta a la etiqueta out1.

```
2068     retval = copy_mount_options(dev_name, &dev_page);
2069     if (retval < 0)
2070          goto out2;
```

Pasa al espacio de direccionamiento del núcleo el nombre del dispositivo a montar. Si se produjera algún error, salta a la etiqueta out2.

```
2072     retval = copy_mount_options(data, &data_page);
2073     if (retval < 0)
2074          goto out3;
```

Pasa al espacio de direccionamiento del núcleo las opciones de montaje. Si se produjera algún error, salta a la etiqueta out3.

```
2076     lock_kernel();
2077     retval = do_mount((char *)dev_page, dir_page, (char *)type_page,
2078               flags, (void *)data_page);
2079     unlock_kernel();
```

Se bloquea el núcleo, se llama a la función do_mount para que realice el montaje y se vuelve a desbloquear el núcleo.

```
2080     free_page(data_page);
2081
2082out3:
2083     free_page(dev_page);
2084out2:
2085     putname(dir_page);
2086out1:
```

```
2087        free_page(type_page);
2088        return retval;
2089}
```

En este punto sólo queda liberar la memoria usada. Si se ha producido algún error, sólo se liberará la memoria reservada hasta ese punto, por esa razón se usan las tres etiquetas.

linux+v2.6.29/fs/namespace.c

do_mount

La function que implementa la operación para montar el sistema también se encuentra en el fichero /fs/namespace.c y es la siguiente:

```
1901long do_mount(char *dev_name, char *dir_name, char *type_page,
1902                unsigned long flags, void *data_page)
1903{
1904        struct path path;
1905        int retval = 0;
1906        int mnt_flags = 0;
1907
1908        /* Discard magic */
1909        if ((flags & MS_MGC_MSK) == MS_MGC_VAL)
1910                flags &= ~MS_MGC_MSK;
1914        if (!dir_name || !*dir_name || !memchr(dir_name, 0, PAGE_SIZE))
1915                return -EINVAL;
1916        if (dev_name && !memchr(dev_name, 0, PAGE_SIZE))
1917                return -EINVAL;
```

En primer lugar, comprueba que el superbloque sea válido; y de no ser así, retorna la macro EINVAL, cuyo significado es precisamente que el superbloque no es válido.

```
1919        if (data_page)
1920                ((char *)data_page)[PAGE_SIZE - 1] = 0;
1921
1922        /* Separate the per-mountpoint flags */
1923        if (flags & MS_NOSUID)
1924                mnt_flags |= MNT_NOSUID;
1925        if (flags & MS_NODEV)
1926                mnt_flags |= MNT_NODEV;
1927        if (flags & MS_NOEXEC)
1928                mnt_flags |= MNT_NOEXEC;
1929        if (flags & MS_NOATIME)
1930                mnt_flags |= MNT_NOATIME;
1931        if (flags & MS_NODIRATIME)
1932                mnt_flags |= MNT_NODIRATIME;
1933        if (flags & MS_RELATIME)
1934                mnt_flags |= MNT_RELATIME;
1935        if (flags & MS_RDONLY)
1936                mnt_flags |= MNT_READONLY;
1937
1938        flags &= ~(MS_NOSUID | MS_NOEXEC | MS_NODEV | MS_ACTIVE |
1939                   MS_NOATIME | MS_NODIRATIME | MS_RELATIME| MS_KERNMOUNT);
```

Se preparan las opciones de montaje en mnt_flag.

```
1942        retval = kern_path(dir_name, LOOKUP_FOLLOW, &path);
1943        if (retval)
1944                return retval;
```

Se obtiene la dirección del punto de montaje. Si se produce algún error en esta llamada, se retorna el correspondiente error devuelto por ***kern_path***.

```
1946        retval = security_sb_mount(dev_name, &path,
1947                                   type_page, flags, data_page);
1948        if (retval)
1949                goto dput_out;
1950
```

Se chequean los permisos antes de montar la unidad.

```
1951        if (flags & MS_REMOUNT)
1952                retval = do_remount(&path, flags & ~MS_REMOUNT, mnt_flags,
1953                                    data_page);
1954        else if (flags & MS_BIND)
1955                retval = do_loopback(&path, dev_name, flags & MS_REC);
1956        else if (flags & (MS_SHARED | MS_PRIVATE | MS_SLAVE | MS_UNBINDABLE))
```

```
1957            retval = do_change_type(&path, flags);
1958    else if (flags & MS_MOVE)
1959            retval = do_move_mount(&path, dev_name);
1960    else
1961            retval = do_new_mount(&path, type_page, flags, mnt_flags,
1962                            dev_name, data_page);
1963dput_out:
1964     path_put(&path);
1965     return retval;
1966}
```

En esta sección de código, se decide que operación debe realizarse dependiendo de los flags activados.
Con la macro **MS_REMOUNT** se comprueba si se desea remontar el sistema de ficheros; si es así, se llama a la función **DO_REMOUNT**, que lo único que hace es modificar las opciones de montaje.
La macro **MS_BIND** se usa para comprobar si se desea hacer que un archivo o un directorio sea visible en otro sistema de fichero. La encargada de ejecutar esta opción es la función **do_loopback.**
Las macros **MS_SHARE, MS_PRIVATE, MS_SLAVE y MS_UNBINDABLE** se usan para comprobar si se desea cambiar a ser compartido, privado, esclavo, in montable. La función que realiza tal acción es **DO_CHANGE_TYPE.**
LA MACRO MS_MOVE SE USA PARA COMPROBAR SI SE DESEA CAMBIAR UN SISTEMA DE ARCHIVOS A UN NUEVO PUNTO DE MONTAJE. LA FUNCIÓN QUE REALIZA TAL ACCIÓN ES DO_MOVE_MOUNT.
Por último, si no se ha incluido ninguna de las opciones anteriores, simplemente se realiza un nuevo montaje con la función **DO_NEW_MOUNT.**

linux+v2.6.29/fs/namespace.c

do_remount

Modifica las opciones del sistema de ficheros montado previamente. El directorio debe contener un punto de montaje principal (raiz de dispositivo).

```
1527/*
1528 * change filesystem flags. dir should be a physical root of filesystem.
1529 * If you've mounted a non-root directory somewhere and want to do remount
1530 * on it - tough luck.
1531 */
1532static int do_remount(struct path *path, int flags, int mnt_flags,
1533                 void *data)
1534{
1535     int err;
1536     struct super_block *sb = path->mnt->mnt_sb;
```

Primero se obtiene información sobre el superbloque.

```
1537
1538     if (!capable(CAP_SYS_ADMIN))
1539             return -EPERM;
1540
1541     if (!check_mnt(path->mnt))
1542             return -EINVAL;
1543
1544     if (path->dentry != path->mnt->mnt_root)
1545             return -EINVAL;
1546
```

Se realizan comprobaciones para poder llevar a cabo las tareas pertinentes. Entre ellas se comprueba que el punto de montaje que se pretende modificar/remontar corresponde a la raíz de un sistema de ficheros.

```
1547     down_write(&sb->s_umount);
1548     if (flags & MS_BIND)
1549             err = change_mount_flags(path->mnt, flags);
1550     else
1551             err = do_remount_sb(sb, flags, data, 0);
1552     if (!err)
1553             path->mnt->mnt_flags = mnt_flags;
1554     up_write(&sb->s_umount);
```

Una vez llegados a este punto, se desactiva la escritura en ese superbloque. Dependiendo de si está activado el flan MS_BIND, es decir, de si ya se encuentra montado (bind); se cambian las opciones de montaje, o se realiza el remontado del superbloque por medio de las funciones ***change_mount_flags*** y ***do_remount_sb***.

```
1555     if (!err) {
1556             security_sb_post_remount(path->mnt, flags, data);
1557
1558             spin_lock(&vfsmount_lock);
1559             touch_mnt_namespace(path->mnt->mnt_ns);
1560             spin_unlock(&vfsmount_lock);
```

```
1561        }
1562        return err;
1563}
```

Por último, si no ha ocurrido ningún error previamente, se cargan los privilegios sobre el punto de montaje y se avisa mediante la función touch_mnt_namespace que el dispositivo se encuentra disponible.

linux+v2.6.29/fs/namespace.c

do_loopback

```
1464static int do_loopback(struct path *path, char *old_name,
1465                                int recurse)
1466{
1467        struct path old_path;
1468        struct vfsmount *mnt = NULL;
1469        int err = mount_is_safe(path);
1470        if (err)
1471                return err;
1472        if (!old_name || !*old_name)
1473                return -EINVAL;
1474        err = kern_path(old_name, LOOKUP_FOLLOW, &old_path);
1475        if (err)
1476                return err;
1477
1478        down_write(&namespace_sem);
1479        err = -EINVAL;
1480        if (IS_MNT_UNBINDABLE(old_path.mnt))
1481                goto out;
1482
1483        if (!check_mnt(path->mnt) || !check_mnt(old_path.mnt))
1484                goto out;
1485
1486        err = -ENOMEM;
1487        if (recurse)
1488                mnt = copy_tree(old_path.mnt, old_path.dentry, 0);
1489        else
1490                mnt = clone_mnt(old_path.mnt, old_path.dentry, 0);
```

Tras hacer una serie de comprobaciones, si se activó previamente el flag BIND, esta función realizará el montaje de un dispositivo ya montado en un nuevo punto de montaje; manteniendo el dispositivo en dos puntos distintos del arbol.

```
1491
1492        if (!mnt)
1493                goto out;
1494
1495        err = graft_tree(mnt, path);
1496        if (err) {
1497                LIST_HEAD(umount_list);
1498                spin_lock(&vfsmount_lock);
1499                umount_tree(mnt, 0, &umount_list);
1500                spin_unlock(&vfsmount_lock);
1501                release_mounts(&umount_list);
1502        }
1503
1504out:
1505        up_write(&namespace_sem);
1506        path_put(&old_path);
1507        return err;
1508}
```

linux+v2.6.29/fs/namespace.c

do_change_type

```
1431static int do_change_type(struct path *path, int flag)
1432{
1433        struct vfsmount *m, *mnt = path->mnt;
1434        int recurse = flag & MS_REC;
1435        int type = flag & ~MS_REC;
1436        int err = 0;
1437
1438        if (!capable(CAP_SYS_ADMIN))
1439                return -EPERM;
1440
1441        if (path->dentry != path->mnt->mnt_root)
1442                return -EINVAL;
1443
1444        down_write(&namespace_sem);
1445        if (type == MS_SHARED) {
1446                err = invent_group_ids(mnt, recurse);
1447                if (err)
1448                        goto out_unlock;
1449        }
1450
1451        spin_lock(&vfsmount_lock);
1452        for (m = mnt; m; m = (recurse ? next_mnt(m, mnt) : NULL))
1453                change_mnt_propagation(m, type);
1454        spin_unlock(&vfsmount_lock);
```

Cambia explícitamente las opciones de los sistemas de ficheros montados, uno por uno.

```
1455
1456 out_unlock:
1457        up_write(&namespace_sem);
1458        return err;
1459}
1460
```

linux+v2.6.29/fs/namespace.c

do_move_mount

```
1575static int do_move_mount(struct path *path, char *old_name)
1576{
1577        struct path old_path, parent_path;
1578        struct vfsmount *p;
1579        int err = 0;
1580        if (!capable(CAP_SYS_ADMIN))
1581                return -EPERM;
1582        if (!old_name || !*old_name)
1583                return -EINVAL;
1584        err = kern_path(old_name, LOOKUP_FOLLOW, &old_path);
1585        if (err)
1586                return err;
1588        down_write(&namespace_sem);
1589        while (d_mountpoint(path->dentry) &&
1590               follow_down(&path->mnt, &path->dentry))
1591                ;
1592        err = -EINVAL;
1593        if (!check_mnt(path->mnt) || !check_mnt(old_path.mnt))
1594                goto out;
1596        err = -ENOENT;
1597        mutex_lock(&path->dentry->d_inode->i_mutex);
1598        if (IS_DEADDIR(path->dentry->d_inode))
1599                goto out1;
1601        if (!IS_ROOT(path->dentry) && d_unhashed(path->dentry))
```

```
1602            goto out1;
1604    err = -EINVAL;
1605    if (old_path.dentry != old_path.mnt->mnt_root)
1606            goto out1;
1608    if (old_path.mnt == old_path.mnt->mnt_parent)
1609            goto out1;
1611    if (S_ISDIR(path->dentry->d_inode->i_mode) !=
1612        S_ISDIR(old_path.dentry->d_inode->i_mode))
1613            goto out1;
1617    if (old_path.mnt->mnt_parent &&
1618        IS_MNT_SHARED(old_path.mnt->mnt_parent))
1619            goto out1;
1624    if (IS_MNT_SHARED(path->mnt) &&
1625        tree_contains_unbindable(old_path.mnt))
1626            goto out1;
1627    err = -ELOOP;
1628    for (p = path->mnt; p->mnt_parent != p; p = p->mnt_parent)
1629            if (p == old_path.mnt)
1630                    goto out1;
```

```
1631
1632    err = attach_recursive_mnt(old_path.mnt, path, &parent_path);
1633    if (err)
1634            goto out1;
```

Se hace una búsqueda, que comprueba las restricciones del padre, compartición, etc. Y luego se hace otra búsqueda escalando por el árbol de directorios antes de llamar a la punción attach_recursive_mnt para añadir el dispositivo en la nueva ruta.

```
1638    list_del_init(&old_path.mnt->mnt_expire);
```

Luego se hace un list_del_init para actualizar el tiempo de validez del nuevo punto de montaje. Y en cualquier punto que se salga, se desbloquean los semáforos y se activa la escritura.

```
1639out1:
1640    mutex_unlock(&path->dentry->d_inode->i_mutex);
1641out:
1642    up_write(&namespace_sem);
1643    if (!err)
1644            path_put(&parent_path);
1645    path_put(&old_path);
1646    return err;
1647}
```

linux+v2.6.29/fs/namespace.c

do_new_mount

Se encarga de crear un nuevo montaje y añadirlo en la lista. Esta función se basa en la función DO_KERN_MOUNT y esta a su vez en VFS_KERN_MOUNT que es la función que realmente nos interesa, ya que es la que realiza las acciones necesarias para hacer un montaje.
El código de esta función está definido en el fichero linux/fs/namespace.c, y podemos considerar que son sólo dos las líneas importantes (llamar a DO_KERN_MOUNT y añadir el nuevo sistema montado a la lista, DO_ADD_MOUNT), ya que el resto son meras comprobaciones.

```
1649/*
1650 * create a new mount for userspace and request it to be added into the
1651 * namespace's tree
1652 */
1653static int do_new_mount(struct path *path, char *type, int flags,
1654                        int mnt_flags, char *name, void *data)
1655{
1656    struct vfsmount *mnt;
1657
1658    if (!type || !memchr(type, 0, PAGE_SIZE))
1659            return -EINVAL;
1660
1661    /* we need capabilities... */
1662    if (!capable(CAP_SYS_ADMIN))
1663            return -EPERM;
1664
1665    mnt = do_kern_mount(type, flags, name, data);
1666    if (IS_ERR(mnt))
```

```
1667            return PTR_ERR(mnt);
1668
1669    return do_add_mount(mnt, path, mnt_flags, NULL);
1670}
```

linux+v2.6.29/fs/namespace.c

La function do_kern_mount se encarga de hacer unas comprobaciones sobre la existencia del tipo de sistema de ficheros; devolviendo el valor de la macro ENODEV con valor 19 avisando que el dispositivo no ha sido encontrado.

```
 993struct vfsmount *
 994do_kern_mount(const char *fstype, int flags, const char *name, void *data)
 995{
 996    struct file_system_type *type = get_fs_type(fstype);
 997    struct vfsmount *mnt;
 998    if (!type)
 999        return ERR_PTR(-ENODEV);
1000    mnt = vfs_kern_mount(type, flags, name, data);
1001    if (!IS_ERR(mnt) && (type->fs_flags & FS_HAS_SUBTYPE) &&
1002        !mnt->mnt_sb->s_subtype)
1003        mnt = fs_set_subtype(mnt, fstype);
1004    put_filesystem(type);
1005    return mnt;
1006}
```

linux+v2.6.29/fs/super.c

Macro devuelta por do_kern_mount:

```
 22#define ENODEV      19    /* No such device */
```

linux+v2.6.29/include/asm-generic/errno-base.h

vfs_kern_mount

La function **vfs_kern_mount** es quizás la que más nos interesa, ya que realiza la mayor parte del trabajo del mount. Es precisamente esta función la encargada de rellenar la estructura ***vfsmount***.

```
 917struct vfsmount *
 918vfs_kern_mount(struct file_system_type *type, int flags, const char *name, void *data)
 919{
 920    struct vfsmount *mnt;
 921    char *secdata = NULL;
 922    int error;
 923
 924    if (!type)
 925        return ERR_PTR(-ENODEV);
 926
 927    error = -ENOMEM;
 928    mnt = alloc_vfsmnt(name);
 929    if (!mnt)
 930        goto out;
```

En primer lugar, la función intenta reservar espacio para la nueva estructura ***vsfmount***, terminando (goto out) en caso que no se obtuviese.

```
 931
 932    if (data && !(type->fs_flags & FS_BINARY_MOUNTDATA)) {
 933        secdata = alloc_secdata();
 934        if (!secdata)
 935            goto out_mnt;
 936
 937        error = security_sb_copy_data(data, secdata);
 938        if (error)
 939            goto out_free_secdata;
 940    }
 941
 942    error = type->get_sb(type, flags, name, data, mnt);
 943    if (error < 0)
 944        goto out_free_secdata;
 945    BUG_ON(!mnt->mnt_sb);
```

```
946
947     error = security_sb_kern_mount(mnt->mnt_sb, flags, secdata);
948     if (error)
949         goto out_sb;
950
```

Obtiene la información necesaria del superbloque para poder rellenar más tarde la estructura y realiza diversas comprobaciones; desviando el flujo de ejecución (goto out_...) en caso que ocurriera algún error.

```
951     mnt->mnt_mountpoint = mnt->mnt_root;
952     mnt->mnt_parent = mnt;
953     up_write(&mnt->mnt_sb->s_umount);
954     free_secdata(secdata);
955     return mnt;
```

Si todo ha ido bien, se rellenan los campos de la estructura ***vfsmount***; y como detalle debemos observar que el dispositivo que se está montando, en su campo parent, apunta a la propia estructura.

```
956out_sb:
957     dput(mnt->mnt_root);
958     up_write(&mnt->mnt_sb->s_umount);
959     deactivate_super(mnt->mnt_sb);
960out_free_secdata:
961     free_secdata(secdata);
962out_mnt:
963     free_vfsmnt(mnt);
964out:
965     return ERR_PTR(error);
966}
```

Si no se retornó en un punto anterior, significa que algo no ha ido bien. Se libera la memoria de las estructuras previamente declaradas según su punto de salida (goto), y se devuelve un parámetro de error.

linux+v2.6.29/fs/super.c

Umount

La orden umount despega (desmonta) de la jerarquía o árbol de ficheros el sistema de ficheros mencionado. Un sistema de ficheros se puede especificar bien dando el directorio donde ha sido montado, o bien dando el dispositivo o fichero especial donde reside. Un sistema de ficheros no puede desmontarse cuando está `ocupado': por ejemplo, cuando hay en él ficheros abiertos, o cuando algún proceso tiene su directorio de trabajo allí, o cuando un fichero de trasiego en él está en uso. El proceso que impide el desmontaje podría ser incluso el mismo umount: él abre libc, y libc a su vez puede abrir por ejemplo ficheros de localización.

Las opciones para la orden umount son:

- **-V** Muestra el número de versión y acaba.
- **-h** Muestra un mensaje de ayuda y acaba.
- **-v** Modo prolijo.
- **-n** Desmonta sin escribir en /etc/mtab.
- **-r** En el caso de que el desmontaje falle, intenta re-montar de lectura exclusiva.
- **-a** Se desmontan todos los sistemas de ficheros descritos en /etc/mtab. (Con las versiones 2.7 de umount y superiores: el sistema de ficheros proc no se desmonta.)
- **-t *tipofsv*** Indica que las acciones sólo deben efectuarse sobre sistemas de ficheros del tipo especificado. Se puede dar más de un tipo empleando una lista de tipos separados por comas. La lista de tipos de sistemas de ficheros puede llevar como prefijo la partícula no para especificar los tipos de sistemas de ficheros sobre los cuales no se tomará ninguna acción.

En caso de error, la variable ERRNO puede tomar los siguientes valores:

ERROR	DESCRIPCIÓN
EBUSY	El sistema de ficheros especificado contiene ficheros abiertos.
EFAULT	specialfile o dir contienen una dirección incorrecta.
ENAMETOOLONG	specialfile o dir especifican un nombre de fichero que es demasiado largo.

ELOOP	Un bucle de un link simbólico se ha encontrado.
ENOENT	specialfile o dir hacen referencia a un nombre de fichero no existente.
ENOMEM	El núcleo no puede asignar memoria para ese descriptor interno.
ENTBLK	specialfile no especifica un nombre de fichero especial.
ENOTDIR	Uno de los componentes de specialfile o dir, usados como nombre de directorio, no es un directorio, o dir no especifica el nombre de un directorio.
EPERM	El proceso no posee los privilegios necesarios.

Llamada al sistema umount

Tal y como comentamos en la llamada al sistema mount, no se encuentra definida como tal, sino a través de una función genérica a la que se pasa el parámetro umount para realizar su función. Su código es el siguiente:

```
1133SYSCALL_DEFINE2(umount, char __user *, name, int, flags)
1134{
1135        struct path path;
1136        int retval;
1137
1138        retval = user_path(name, &path);
1139        if (retval)
1140                goto out;
```

Por medio de la función **user_path** se comprueba si el usuario mantiene ocupado el dispositivo. Siguiendo la declaración de esta función comprobamos que es un cambio con respecto a las anteriores. Esta función "user_path" está descrita como una macro que hace referencia a **user_path_at** en el fichero linux+v2.6.29/fs/namei.c

```
1141        retval = -EINVAL;
1142        if (path.dentry != path.mnt->mnt_root)
1143                goto dput_and_out;
1144        if (!check_mnt(path.mnt))
1145                goto dput_and_out;
```

Comprueba que el punto de montaje sobre el que se está trabajando es válido.

```
1147        retval = -EPERM;
1148        if (!capable(CAP_SYS_ADMIN))
1149                goto dput_and_out;
```

Se comprueba que el proceso que hizo la llamada al sistema tiene los derechos necesarios para realizarla.

```
1151        retval = do_umount(path.mnt, flags);
```

Esta función es la que realmente realiza la tarea de desmontar el dispositivo.

```
1152dput_and_out:
1153        /* we mustn't call path_put() as that would clear mnt_expiry_mark */
1154        dput(path.dentry);
1155        mntput_no_expire(path.mnt);
1156out:
1157        return retval;
1158}
```

Por ultimo, vemos que se puede llegar a la sección dput_and_out desde el chequeo de permisos, o mediante el flujo normal de ejecución tras superar do_umount. En esta parte se desvincula el punto de montaje que fue indicado. En caso que salte esta sección (goto) se devolvería el correspondiente error.

linux+v2.6.29/fs/namespace.c

Función do_umount

Esta función se encuentra en el fichero linux/fs/namespace.c

```
1036static int do_umount(struct vfsmount *mnt, int flags)
1037{
```

```
1038    struct super_block *sb = mnt->mnt_sb;
1039    int retval;
1040    LIST_HEAD(umount_list);
1042    retval = security_sb_umount(mnt, flags);
1043    if (retval)
1044        return retval;
```

Activa opciones de seguridad antes de desmontar el dispositivo, pasandole a la función ***security_sb_umount*** los parámetros (flags) pertinentes.

```
1052    if (flags & MNT_EXPIRE) {
1053        if (mnt == current->fs->root.mnt ||
1054            flags & (MNT_FORCE | MNT_DETACH))
1055            return -EINVAL;
1057        if (atomic_read(&mnt->mnt_count) != 2)
1058            return -EBUSY;
1060        if (!xchg(&mnt->mnt_expiry_mark, 1))
1061            return -EAGAIN;
1062    }
```

En esta sección se observa si se activó el flag MNT_EXPIRE, y demás condiciones; para permitir al usuario a pedir que un dispositivo sea desmontado y exista interacción en vez de desmontarlo incondicionalmente.

```
1074    if (flags & MNT_FORCE && sb->s_op->umount_begin) {
1075        lock_kernel();
1076        sb->s_op->umount_begin(sb);
1077        unlock_kernel();
1078    }
```

Cuando se hace una petición de desmontado forzado de una unidad, se realiza la operación bloqueando el núcleo; de manera que se deja continuar en caso de un posible fallo (sin retornar). Esto se hace siguiendo la filosofía que si algo falla sea informado posteriormente en el espacio de usuario y se deje continuar al sistema. En caso que se requiera forzar la operación umount, se chequea si está activo el flan ***MNT_FORCE*** invocando a la operación umount_begin de entre las operaciones del superbloque. Esta operación existe además con el propósito de poder desmontar el sistema de ficheros en caso que un usuario retire el recurso antes de haberse desmontado.

```
1089    if (mnt == current->fs->root.mnt && !(flags & MNT_DETACH)) {
1094        down_write(&sb->s_umount);
1095        if (!(sb->s_flags & MS_RDONLY)) {
1096            lock_kernel();
1097            retval = do_remount_sb(sb, MS_RDONLY, NULL, 0);
1098            unlock_kernel();
1099        }
1100        up_write(&sb->s_umount);
1101        return retval;
1102    }
```

Esta parte del código anteriormente se encargaba de volver a montar los sistemas de fichero en caso que se pidiera expresamente, únicamente cambiando las opciones de montaje. En esta versión ha sido retocada para que contemple la posibilidad de cerrar todos los descriptores e implementar un proceso de rearranque. Existe una gran cantidad de código que varía sensiblemente con respecto a las implementaciones anteriores. En esta sección se repasan los flags ***MNT_DETACH*** y ***MNT_RDONLY*** modificando las opciones del punto de montaje con la función ***do_remount_sb***. Podemos encontrar una nota explicativa haciendo ver que no es una implementación definitiva y admite sugerencias.

```
1104    down_write(&namespace_sem);
1105    spin_lock(&vfsmount_lock);
1106    event++;
1108    if (!(flags & MNT_DETACH))
1109        shrink_submounts(mnt, &umount_list);
1111    retval = -EBUSY;
1112    if (flags & MNT_DETACH || !propagate_mount_busy(mnt, 2)) {
1113        if (!list_empty(&mnt->mnt_list))
1114            umount_tree(mnt, 1, &umount_list);
1115        retval = 0;
1116    }
1117    spin_unlock(&vfsmount_lock);
1118    if (retval)
1119        security_sb_umount_busy(mnt);
1120    up_write(&namespace_sem);
1121    release_mounts(&umount_list);
1122    return retval;
1123}
```

Se bloquean los semáforos para tratar la lista de dispositivos. Se chequea si existe más de un dispositivo montado bajo el mismo punto. Si no es posible desmontar por quedar abierto algún fichero modifica las opciones de seguridad, y se libera el recurso desvinculándolo del árbol y quitándolo de la lista.

linux+v2.6.29/fs/namespace.c