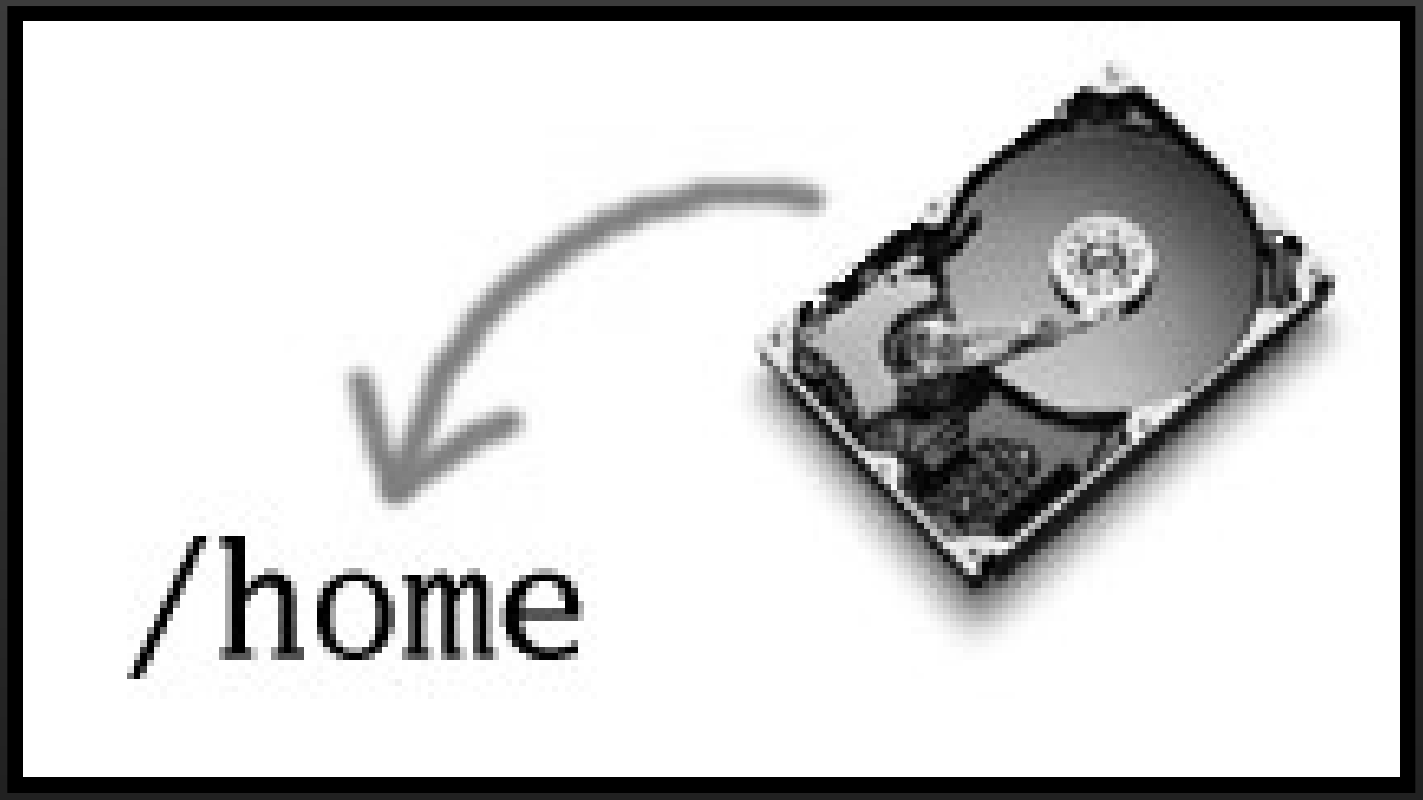
Mount

Un sistema de ficheros son los métodos y las estructuras de datos que emplea el sistema operativo (en nuestro caso, Linux) para organizar los ficheros en disco. El término Sistema de Ficheros se utiliza tanto para referirse a una partición o a un disco.



Intro

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.

- **EXT3:** es una modificación del ext2 para añadirle funcionalidades de journaling.

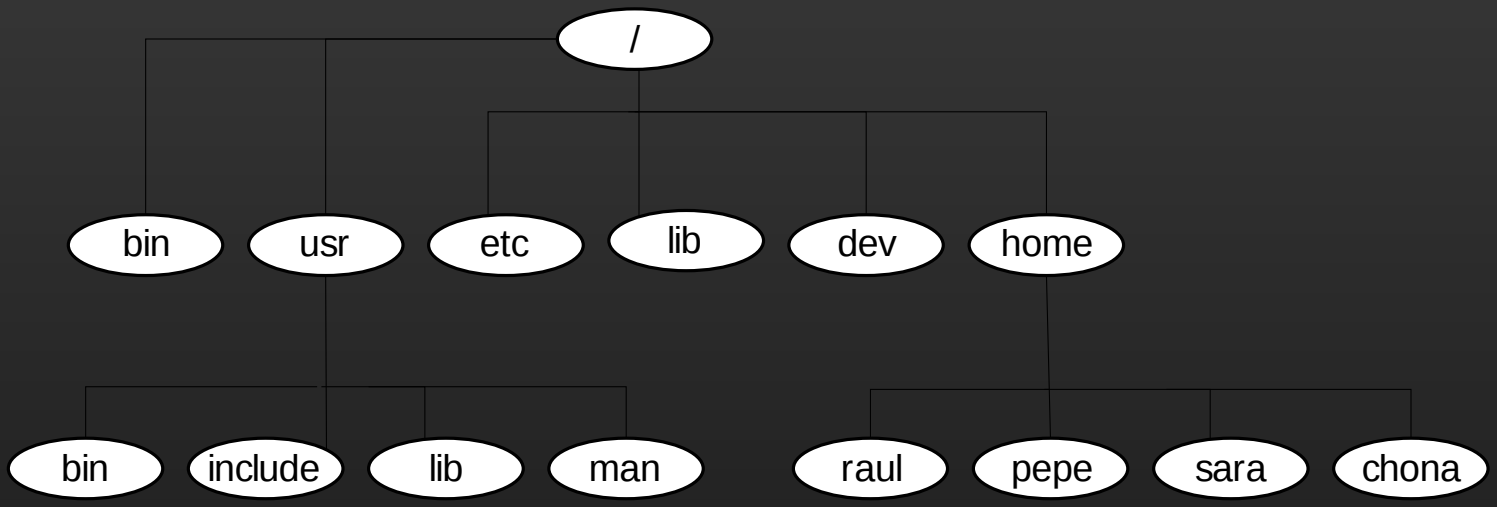
- **VFAT:** este tipo permite utilizar sistemas de ficheros de Windows (FAT, FAT32), y actualmente está soportado el sistema de ficheros de Windows NT, pero sólo fiable en sólo-lectura.

- **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.

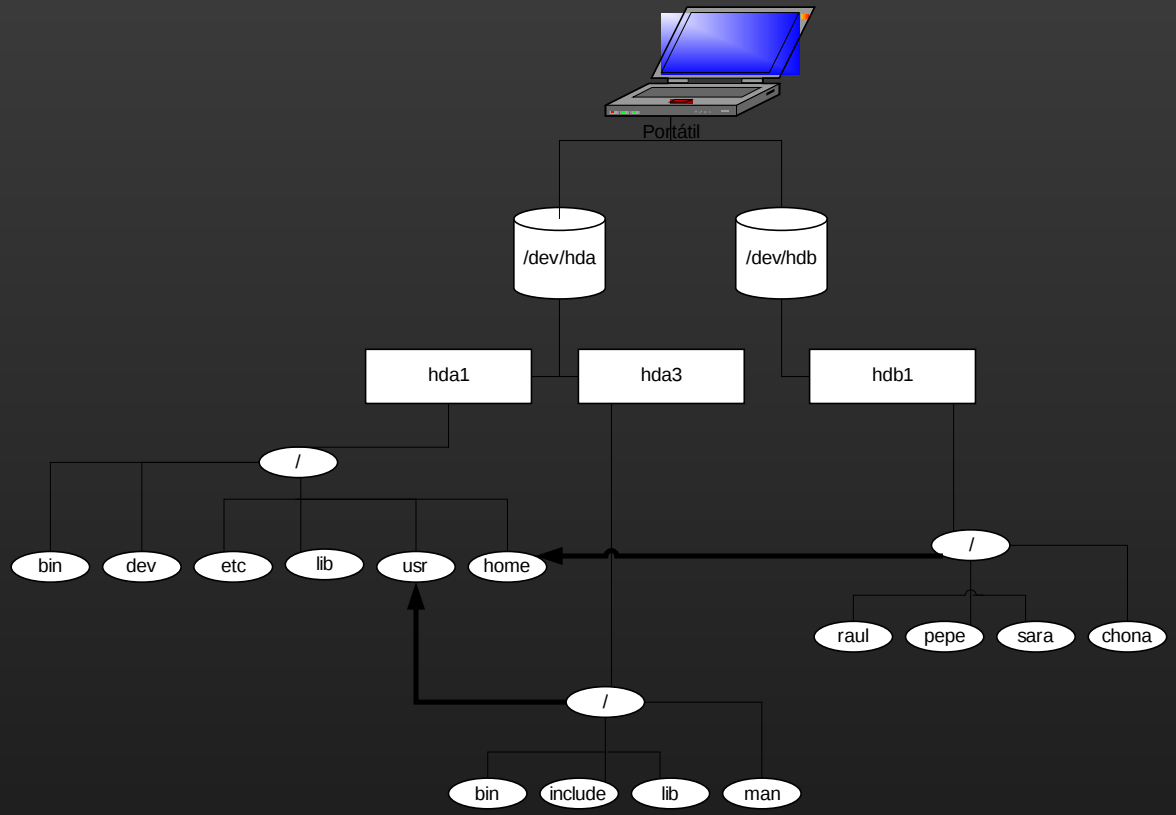
Intro

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.



Intro

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.



Intro

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

Intro

mount --bind/--move

Como 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.

Intro

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.

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

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

```
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

Intro

```
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.
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

Estructura

```
 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;
 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 */
 63      atomic_t mnt_count;
 64        int mnt_expiry_mark;                  /* true if marked for expiry */
 65        int mnt_pinned;
 66        int mnt_ghosts;
 71      atomic_t __mnt_writers;
 72};
```

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

Estructura

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.

sys_mount

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{
```

NOTA: Cambio en la definición de la llamada al sistema

Ahora se hace por medio de una macro que convierte el parámetro de la llamada en el nombre de la función. Esta macro se encuentra definida en el archivo linux+v2.6.29/include/linux/syscalls.h

```
#define SYSCALL_DEFINE5(name, ...) SYSCALL_DEFINEx(5, _##name, __VA_ARGS__)
```

sys_mount

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

1. 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.

3. 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.

5. 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.

7. Y por último, libera la memoria usada para las opciones.

sys_mount


```
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.

sys_mount

```
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 kernel, se llama a la función do_mount para que realice el montaje y se vuelve a desbloquear el kernel.

```
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.

do_mount

```
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.

do_mount

```
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.

do_mount

```
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.

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.

```
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;

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

do_kernel_mount

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

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.

vfs_kern_mount


```
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.

vfs_kern_mount

```
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.

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

umount

La orden umount despega 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.

```
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

umount

```
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.

umount

```
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.