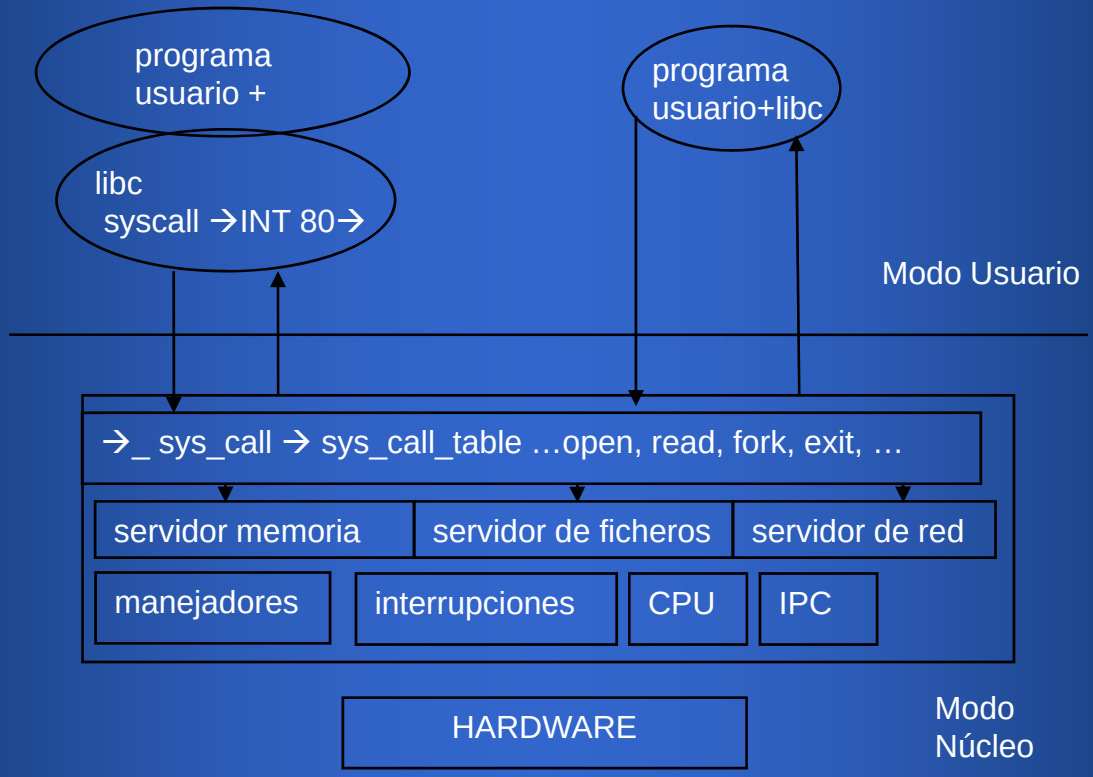
Llamadas al Sistema

s. candela

26.1 Introducción

La interfase entre los programas de usuario y el sistema operativo se define mediante un conjunto de "instrucciones ampliadas" o llamadas al sistema, Linux proporciona unas llamadas definidas en el sistema por un nombre “open”, “read”, “mmap”,….



26.1 Introducción

El conjunto de llamadas que ofrece LINUX, son muy parecidas a las de UNIX y XENIX, si bien estos tienen alguna distinta. LINUX tiene alrededor de 200 llamadas al sistema, idénticas a UNIX V7 (estándar POSIX). en nombre, función y parámetros.

Existe unas librerías de procedimientos en /usr/lib/ para poder realizar llamadas desde un programa escrito en C.

Ejemplo de llamada:

count = **read**(file, buffer, nbyte);

read -> nombre de la llamada al sistema
file -> fichero de donde leer
buffer -> zona de memoria donde colocar los dato
nbytes -> nº de bytes a leer
count -> nº de bytes leídos, si count es - 1, hay error, que se coloca en la variable global errno

26.1 Introducción

Veamos un ejemplo de uso del open para abrir el fichero especial "/dev/tty" asociado con la consola asociada al proceso

```
# include  <fcntl.h>

int main (int argc, char **argv)  {
/* abrimos la consola */
        int fd = open("/dev/tty", O_RDWR);
/* el descriptor de fichero "fd" devuelto por open es
utilizado por write para escribir */
        write(fd, "hola joven!\n", 12);
/* cerramos el fichero */
        exit(0);
}
```

26.1 Introducción

Los programas deben siempre comprobar después de una llamada si todo es correcto, para ello Linux proporciona una variable **errno** y una función **perror()**, ya definidas por el sistema.

Ejemplo de programa, del uso de la variable global **errno** y del procedimiento **perror()**

```
/*  errores.c
*  lista los 53 primeros errores de llamadas al sistema
*/
# include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
int i;
extern int errno;
  for (i=0;i<=53;i++){
      fprintf(stderr,"%3d",i);
      errno=i;
      perror(" ");
  }
  exit(0);
}
```

26.1 Introducción

brk	**getppid**	**getrusage**	**mremap**	**setpgid**
capget	**Getpgrp**	**getpid**	**munlock**	**setpriority**
capset	**getgid**	**getsid**	**munlockall**	**setregid**
clone2	**geteuid**	gettid	**munmap**	**setresgid**
execve	**geteguid**	**getuid**	**nanosleep**	**setresuid**
exit	**getpgid**	**idle**	personality	setreuid
fork	getpid	madvise	prctl	**setrlimit**
getegid	getppid	mincore	ptrace	**setsid**
geteuid	**getpriority**	mlockall	setfsgid	**setuid**
getguid	**getresgid**	**mlock**	setfsuit	**wait4**
getgroups	**getresuid**	**mmap**	**setgid**	**waitpid**
getpagesaize	**getrlimit**	**mprotect**	**setgroups**	

26.1 Introducción

Manejo de procesos

pid = **fork**() - crea un proceso hijo idéntico al proceso padre.
pid = **waitpid**(pid, &statloc, opts) - espera a que un hijo determinado termine y coge su condición de salida.
s = **wait**(&status) - espera a que un proceso hijo termine y coge su condición de salida devuelve el pid del hijo que termina, llamada antigua.
s = **execve** (name, argv, envp) - sustituye la imagen en memoria de un proceso.
exit(status) - pone fin a la ejecución del proceso y devuelve la condición de salida.
size = **brk** (addr) - fija el tamaño del segmento de datos a (addr).
pid = **getpid**() - devuelve el id del proceso solicitante.
pid = **getpgrp**() - devuelve el id de grupo de procesos del proceso solicitante.
pid = **setsid**() - crea una nueva sesión y asigna el pid del invocador como identificador del grupo de procesos.
s = **ptrace**(req, pid, addr, data) - controla un proceso para depurar.

26.1 Introducción Manejo de Señales

alarm **kill** **pause**	**rt_sigreturn** **rt_sigprocmask** **rt_sigpending**	**rt_sigqueueinfo** **rt_sigsuspend** **sigpending**	**sigaltask** **signal** sigpending	sigqueueinfo **sigreturn** **sigsuspend**
rt_sigaction	**rt_sigtimedwait**	**sigaction**	**sigprocmask**	sigtimedwait

s = **signal**(sig,&función) - define la acción a realizar cuando se recibe una señal.
s = **sigaction**(sig, act, oldact) - controla el estado y manejo de señales.
s = **sigreturn**(&context) - regresa de una señal.
s = **sigprocmask**(how, &set, &old) - examina o cambia la mascara de las señales.
s = **sigpending**(set) - obtiene el conjunto de señales bloqueadas.
s = **sigsuspend**(sigmask) - sustituye la mascara de señales y suspende el proceso.
s = **kill** (pid, sig) - envía una señal a un proceso.
residual = **alarm**(seconds) - planifica o programa una señal SIGALRM después de un cierto tiempo.
s = **pause**() - suspende el solicitante hasta la siguiente señal.

26.1 Introducción Manejo de ficheros

access	**fchown**	**lseek**	**pwrite**	**statfs**
acct	**fcntl**	**lstat**	readahead	**stat**
capget	**fdatasync**	**mkdir**	**readlink**	**symlink**
capset	**flock**	**mknod**	**readv**	**sysfs**
chdir	**fstatfs**	**mount**	**read**	**truncate**
chmod	**fstat**	**newstat**	**readdir**	**umask**
chown	**ftruncate**	**newlstat**	**rename**	**umount**
chroot	**getcwd**	**newfstat**	**rmdir**	**uname**
close	**getdents**	**nfsservctl**	**select**	**unlink**
creat	**ioctl**	**open**	**sendfile**	**ustat**
dup	**ioperm**	**pipe**	**setfsuid**	**writev**
dup2	**iopl**	pivot_root	**setfsgid**	**write**
fchdir	**lchown**	**poll**	**sgetmask**	
fchmod	**link**	**pread**	**ssetmask**	

26.1 Introducción Manejo de ficheros

- fd =**creat** (name, mode) - crea un nuevo fichero o trunca uno existente.
- fd = **mknod** (name, mode, addr) - crea un nodo i normal, especial, o de directorio.
- fd = **open** (file, how, ...) - abre un fichero para lectura, escritura o ambos.
- s = **close** (fd) - cierra un fichero abierto.
- n = **read** (fd, buffer, nbytes) - lee datos de un fichero y los coloca en un buffer.
- n = **write** (fd, buffer, nbytes) - escribe datos a un fichero desde un buffer.
- pos = **lseek** (fd, offset, whence) - mueve el apuntador del fichero.
- s = **stat** (name, &buf) - lee y devuelve el estado de un fichero de su nodo i.
- s = **fstat** (fd, buf) - lee y devuelve el estado de un fichero a partir de su nodo i.
- fd = **dup** (fd) - asigna otro descriptor de fichero para un fichero abierto.
- s = **pipe** (&fd [0]) - crea una tubería.
- s = **ioctl**(fd, request, argp) - realiza operaciones especiales en ficheros especiales.
- s = **access**(name,amode) - verifica los accesos a un fichero.
- s = **rename**(old, new) - cambia el nombre de un fichero.
- s = **fcntl**(fd, cmd, ...) - bloqueo de un fichero y otras operaciones.

26.1 Introducción Manejo del directorio y del sistema de ficheros

- s = **mkdir**(name, mode) - crea un nuevo directorio.
- s = **rmdir**(name) - elimina un directorio vacio.
- s = **link** (name1, name2) - crea una entrada nueva name2 en el directorio, que apunta al fichero name1.
- s = **unlink** (name) - elimina una entrada del directorio.
- s = **mount** (special, name, rwflag) - monta un sistema de ficheros.
- s = **unmount** (special) - desmonta un sistema de ficheros.
- s = **sync**() - escribe todos los bloques de la memoria cache en el disco.
- s = **chdir** (dirname) - cambia el directorio de trabajo.
- s = **chroot** (dirname) - cambia al directorio raíz.

26.1 Introducción Protección

- s = **chmod** (name, mode) - cambia los bits de protección asociados con un fichero.
- iud = **getuid**() - determina el uid del solicitante.
- gid = **getgid**() - determina el gid del solicitante.
- s = **setuid**(uid) - fija el uid del solicitante.
- s = **setgid**(gid) - fija el gid del solicitante.
- s = **chown** (name, owner, group) - cambia el propietario y grupo de un fichero.
- oldmask = **umask** (complmode) - pone una máscara que se utiliza para fijar los bits de protección.

26.1 Introducción Manejo de tiempo

adjtimex	**settimeofday**	**times**
getitimer	**setitimer**	**utime**
gettimeofday	**stimes**	

seconds = **time** (&seconds) - calcula el tiempo transcurrido en segundos desde el 1 de enero de 1970.

s = **stime** (tp) - pone el tiempo transcurrido desde el 1 de enero 1970.

s = **utime** (file, timep) - pone la hora del "último acceso" del fichero.

s = **times** (buffer) - determina los tiempos del usuario y del sistema gastados hasta ahora.

26.1 Introducción Manejo de la CPU -Comunicación entre procesos IPC

Manejo de la CPU

sched_getparam **sched_setparam** **sched_yield**	**sched_get_priority_maxs** **ched_get_priority_minsch** **ed_rr_get_intrerval**	**sched_getscheduler** **sched_setscheduler**

Comunicación entre procesos IPC

ipc msgctl msgget	msgrcv msgsnd semctl	semget semop	shmat shmctl	shmdt shmget

26.1 Introducción Socket - Miscelaneos

- **Socket**

accept bind connect getpeername	getsocknam e getsockpt listen recvfrom	recvmsg recv sendmsg sendto	send setsockopt shutdown socketpair	socket **socketcall**

- **Miscelaneos**

bdflush **create_module** **delete_module** fdatasync **fsync** **get_kernel_syms**	**init_module** **modify_ldt** **msync** pciconfig_read pciconfig_write perfmonctl	**prctl** **Ptrace** **query_module** **quotactl** **reboot** **setdomainname**	**sethostname** **swapoff** **swapon** **sync** **sysctl** sysfs	**sysinfo** **syslog** **syspersonality** **uname** **vhangup** **uselib**

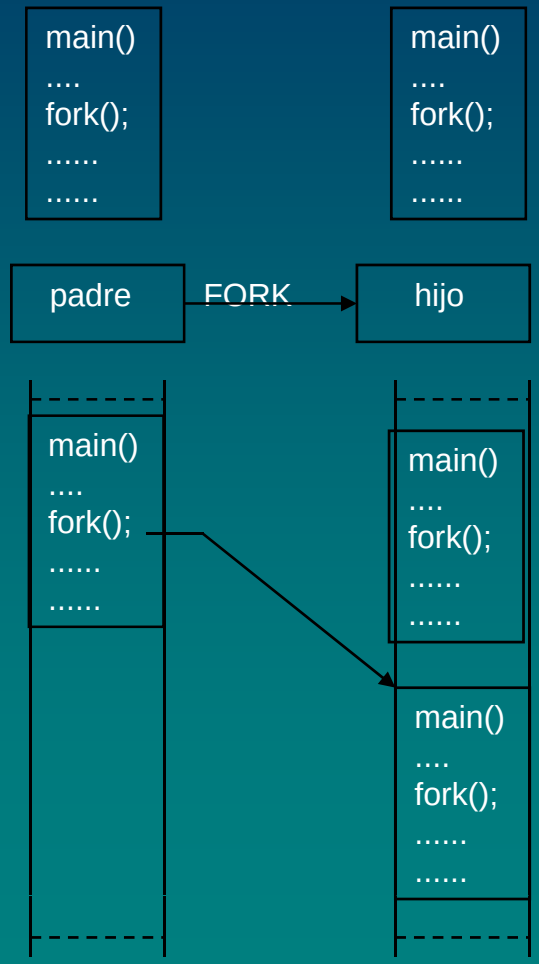
26.2 Llamadas manejo de procesos

FORK()

- Crea un nuevo proceso, exactamente igual al proceso original, incluyendo los descriptores de ficheros, registros, ambiente, etc.
- Una vez ejecutado el FORK, padre, e hijo se están ejecutando simultáneamente, en un principio las variables tienen los mismos valores, después, cambios en uno no afectan al otro.
- La llamada FORK devuelve un valor después de ejecutarse, este valor es:
  - cero -> en el proceso hijo
  - pid (identificador del proceso hijo -> en el padre
- Mediante este valor los procesos pueden saber cual es el padre y cual es el hijo.

26.2 Llamadas manejo de procesos

FORK()



26.2 Llamadas manejo de procesos

WAITPID

- pid = **waitpid(pid, &statloc, opts)** - espera a que un hijo determinado termine, devolviendo el pid del proceso que termina, y coge su condición de salida.
- En mucho casos, el hijo creado por el FORK necesita lanzar un proceso, distinto al del padre. Ejemplo:
- El Shell, lee un comando, lanza un proceso hijo, y debe esperar hasta que el proceso hijo ejecute el comando, posteriormente lee un nuevo comando cuando el hijo termina.
- Para esperar a que el hijo termine, el padre debe ejecutar WAITPID.
- Si el parámetro **pid** vale -1, el padre espera por el primer hijo que termina
- El segundo parámetro **statloc** es la dirección de una variable (estado de salida) que es escrita por el proceso hijo para indicar terminación normal o anormal.

26.2 Llamadas manejo de procesos

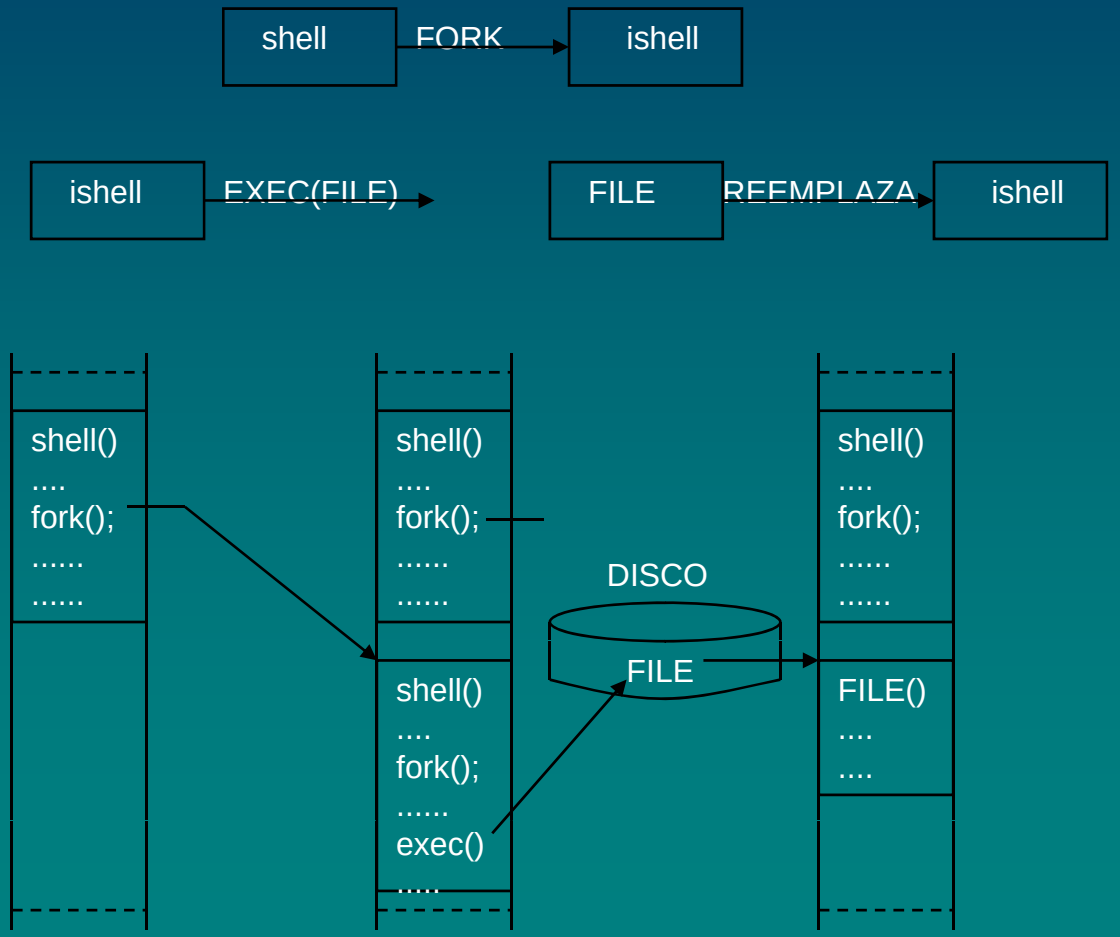
EXEC

- s = **execve** (name, argv, envp) - sustituye la imagen en memoria de un proceso.
- name - nombre del fichero que contiene el programa a ejecutar.
- argv - vector de argumentos.
- envp - vector ambiente

- En el caso del Shell, el proceso hijo debe ejecutar el comando, esto lo hace mediante la llamada EXEC, la cual hace que la imagen del hijo, sea reemplazada por el fichero que hay en el primer parámetro de EXEC.

26.2 Llamadas manejo de procesos

EXEC



26.2 Llamadas manejo de procesos

EXEC

Ejemplo: Shell simplificado con FORK, WAITPID y EXEC.

```
While (TRUE){                          /* bucle indefinido */
read_command (command, parámetro);     /* lee comando  */
if ( fork() != 0){                     /* lanza un proceso hijo */
   waitpid(-1,&status,0);              /* el padre espera a que hijo acabe */
   }else{
   execve(command, parámetros, 0);     /* ejecuta el comando */
  }
}
```

Pregunta: ¿El Shell padre podría lanzar directamente el comando?

La respuesta es NO, ya que EXEC, reemplaza el fichero por el código de quien lo lanza, y este se pierde.

Muchas librerías, contienen varias versiones del EXEC.

execl, execv, execle, execve

que permiten omitir o especificar los parámetros de forma particular. Utilizaremos **EXEC** para referirnos a cualquiera de ellos.

26.2 Llamadas manejo de procesos

EXEC

Consideremos el ejemplo: **cp file1 file2** copiar file1 en file2

Después que el Shell ha creado con el FORK una imagen hijo, este hijo ejecuta cp mediante EXEC, le pasa al programa cp, y la información a cerca de los ficheros a copiarse.

El programa principal main de cp contiene los argumentos: main (argc, argv, envp)

argc -> Entero, contiene el número de parámetros (palabras) contenidos en la línea de comandos, en nuestro ejemplo 3.

argv -> Es un puntero a un vector, cada elemento del vector contiene un parámetro de la línea de comandos.

argv[0] = cp
argv[1] = file1
argv[2] = file2

envp -> Es un pointer o un vector ambiente, es un vector estring, conteniendo asignaciones de la forma (nombre = valor), utilizada para pasar información como el tipo del terminal, directorio, símbolo del Shell, etc.

26.2 Llamadas manejo de procesos

EXIT().

Esta llamada la deben utilizar lo procesos cuando terminan.

Devuelve al padre un valor de estatus (0 a 255), que se carga en el byte menos significativo de la variable estatus de WAITPID, un cero es terminación normal, otro valor es error.

BRK(adr)

Esta llamada cambia el tamaño del segmento de datos de un programa. El parámetro **adr** que indica el nuevo tamaño en bytes del segmento de datos. Devuelve el tamaño actual del segmento de datos.
La rutina **sbrk**, en la librería, también cambia el tamaño del segmento de datos, su parámetro indica el número de bytes que aumenta (positivo) o disminuye (negativo) el segmento de datos.

GETPID

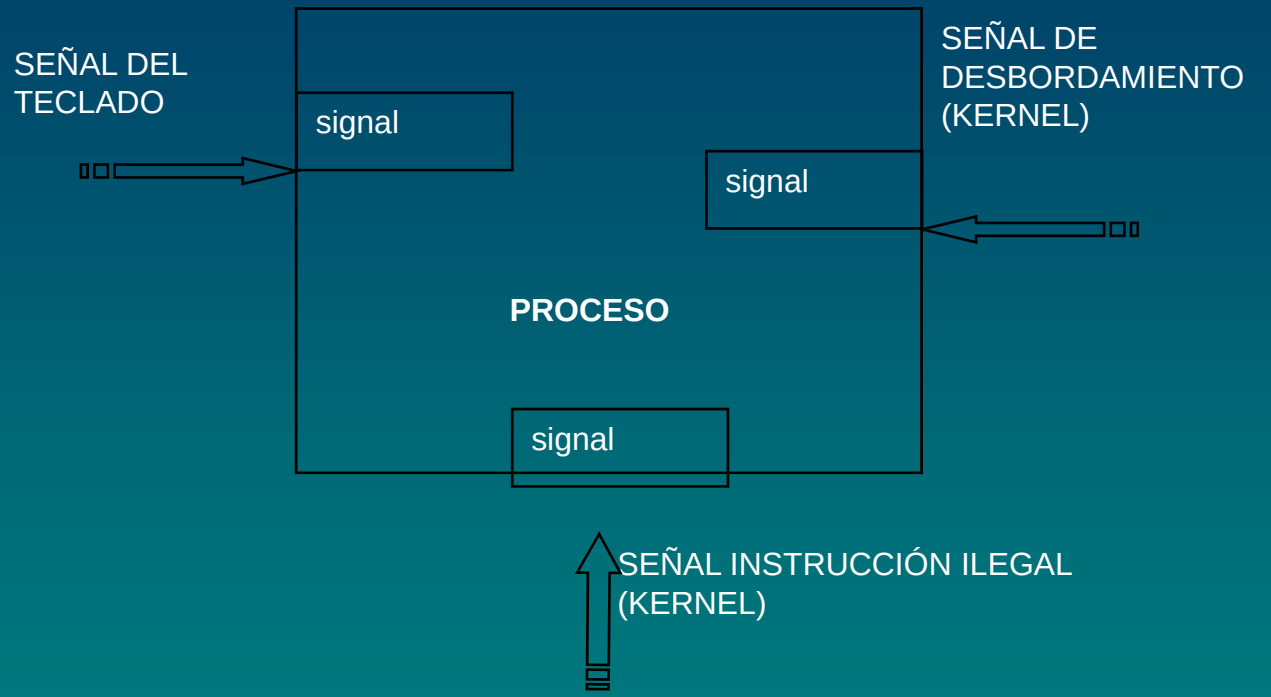
Devuelve el identificador del proceso (pid) que lo ejecuta.
En un Fork, al padre se le devuelve el pid del hijo, mientras que al hijo se le devuelve un cero, si el hijo quiere saber su pid, deberá efectuar un GETPID.

26.3 Llamadas manejo de señales

Las llamadas se utilizan para
manejar interrupciones por software
hacer reports de interrupciones hardware
detectar instrucciones ilegales
rebosamiento de variables
Cuando una señal se envía a un proceso, que no esté preparado para recibirla, el proceso se mata.
SIGNAL(señal, &funcion)
    int señal;
    int (*función)(); función a realizar cuando se recibe una señal

Esta llamada se utiliza para indicar que se está preparado para recibir algún tipo de señal y suministrar la dirección del procedimiento manejador de la señal.

26.3 Llamadas manejo de señales



Después del SIGNAL, si se recibe una señal por ejemplo SIGINT (pulsar la tecla DEL), se efectúa:

1) El estado del proceso se guarda en su propia pila.
2) Se ejecuta el manejador de la señal.

26.3 Llamadas manejo de señales

Los tipos de señales son:

1 SIGHUP El modem ha detectado línea telefónica colgada
2 SIGINT Interrupción de teclado, la tecla DEL ha sido pulsada
3 SIGQUIT Señal Quit (desde el teclado CTRL \)
4 SIGILL Instrucción ilegal
5 SIGTRAP Traza de los traps (excepciones)
6 SIGABRT Abortar un programa
7 SIGBUS Error en el Bus
8 SIGFPE Excepción por rebosamiento de coma flotante
9 SIGKILL Matar un proceso
10 SIGUSR1 El usuario define la señal # 1
11 SIGSEGV Violación de segmentación
12 SIGUSR2 El usuario define la señal # 2
13 SIGPIPE Escritura en pipe sin lectores
14 SIGALRM Alarma
15 SIGTERM Software genera una señal de terminación
16 SIGSTKFLT Fallo en el stack
17 SIGCHLD Cambia el estado del hijo, terminado o bloqueado

26.3 Llamadas manejo de señales

Los tipos de señales son:

18 SIGCONT Si está bloqueado continuar
19 SIGSTOP Señal de paro
20 SIGTSTP Paro del teclado
21 SIGTTIN Proceso en segundo plano quiere leer, lectura terminal
22 SIGTTOU Proceso en segundo plano quiere escribir, escritura terminal
23 SIGURG Condición urgente
24 SIGXCPU Excedido el límite de la CPU
25 SIGXFSZ Excedido el limite del tamaño de un fichero
26 SIGVTALRM Alarma del reloj virtual
27 SIGPROF Historial del reloj
28 SIGWINCH Cambia el tamaño de la ventana
29 SIGIO Ahora es posible la entrada salida
30 SIGPWR Fallo en la alimentación
31 SIGSYS Llamada al sistema erronea
32 SIGRTMIN Primera señal en tiempo real
33 SIGRTMAX Última señal en tiempo real

26.3 Llamadas manejo de señales

Después de recibir una señal, en principio es necesario volver a permitir recibir otra señal con otro SIGNAL, ya que si no, puede recibirse una señal y matar el proceso (acción por defecto), existe un parámetro para que se vuelva a cargar la función.

Existen dos funciones predefinidas que el usuario puede utilizar:
SIG_IGN, para que las señales se ignoren (excepto SIGKILL).
SIG_DFL para ejecutar la acción por defecto de matar al proceso.

Ejemplo: Supongamos que lanzamos un comando en modo tanda:
command &

Es indeseable que la señal DEL, del teclado pueda afectar a ese proceso, así, el Shell después de ejecutar el FORK, pero antes del EXEC deberá hacer
signal (SIGNIT, SIG_IGN);
signal (SIGQUIT, SIG_IGN);
que inhiben las señales DEL, y QUIT (CTRL\).

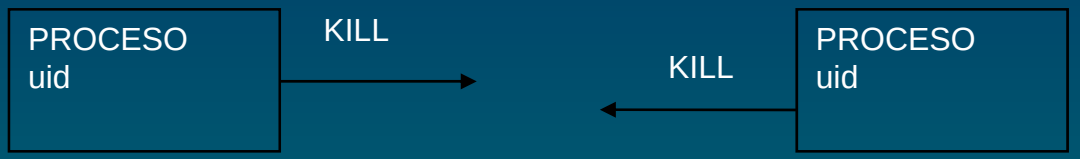
26.3 Llamadas manejo de señales

SIGACTION (sig, *act, *oact)

Es una nueva versión de signal, examina, pone o modifica los atributos de una señal.
Sig es un entero que indica la señal.
*act es una estructura que contiene los atributos y manejador de la señal
*oact es la estructura que recibe los atributos antes de la llamada.

26.3 Llamadas manejo de señales

KILL



La llamada kill, permite enviar señales entre procesos que tienen el mismo uid (identificador de usuario) la forma es: kill (pid, sig)

pid -> identificador del proceso
sig -> tipo de señal

sig = 9 es la señal SIGKILL, que mata cualquier proceso, aún en modo tanda, esta señal no puede ser ignorada.

26.3 Llamadas manejo de señales

ALARM (segundos)

Esta llamada se utiliza para interrumpir un proceso al cabo de un cierto tiempo, su parámetro especifica el tiempo en segundos, después del cual el núcleo envía la señal SIGALRM al proceso.
Un proceso solo puede tener una alarma pendiente, para resetear una alarma, basta hacer una llamada de alarma con parámetro cero.

PAUSE (segundos)

Esta llamada le dice al S.O. que suspenda al proceso, hasta que se reciba una señal.

26.3 Llamadas manejo de señales

SIGPROCMASK(how, &set, &old)

Esta señal permite bloquear un conjunto de señales mediante una mascara de bits que le envía al núcleo del sistema.

SIGPENDING(SET)

Obtiene el conjunto de señales bloqueadas.

SIGSUSPEND(sigmask)

Permite a un proceso establecer atómicamente el mapa de bits de las señales bloqueadas y suspenderse a sí mismo.

26.4 Llamadas manejo de ficheros

CREAT

Crea un fichero y lo abre en modo escritura, independiente del modo. Sus parámetros son el nombre del fichero y el permiso.

```
int creat (nombre, permisos)       /* crea un fichero */
    char *nombre;                  /* nombre camino */
    int permisos;                  /* bits de permisos */
```

fd = CREAT("ac", 0751)

Crea un fichero de nombre **ac** y con permiso **0751**.

Permiso está escrito en octal, cada dígito tiene tres bits que se relacionan con los modos rwx, lectura, escritura y ejecutar, un uno significa permiso, cero no permiso.

El tercer dígito (7) se refiere al propietario rwx todo.

El segundo dígito (5) a los usurarios del grupo r-x leer y ejecutar.

El primer dígito (1) al resto --x sólo ejecutar.

Si el fichero ya existe, el fichero se trunca a longitud cero, si el permiso es correcto.

Esta llamada nos devuelve el fd (descriptor del fichero) que se utiliza para posteriores referencias al fichero.

26.4 Llamadas manejo de ficheros

MKNOD

```
int mknod (nombre, modo, equipo)  /* crea un fichero especial */
char *nombre;          /* fichero a crearse */
int modo;              /* modo del nuevo fichero */
int equipo;            /* numero de equipo */
```

Crea un fichero especial, es solo llamable por el super usuario.

fd = mknod ("/dev/tty2", 020744, 0x0402);

- Crea un fichero llamado /dev/tty2 (nombre usual para el terminal 2).
- Modo en octal 020744, indica que es un fichero especial en modo carácter c..rwxr--r--
- El tercer parámetro expresa en el byte más significativo 04 (mayor equipo) el tipo de equipo (disco, terminal impresora, etc) y en el byte menos significativo (menor equipo) el número dentro de su tipo, en este caso 02.

26.4 Llamadas manejo de ficheros

OPEN

Abre un fichero o crea un fichero, forma moderna de creat.

```
int open (fichero, flags, modo)        /* abre un fichero */
char *fichero;                  /* nombre camino  */
int flags;                      /* lectura escritura o ambos */
Mode_t modo                     /* especifica los permisos fichero
(modificado por umask) */
```

flags

Abre un fichero, declarando un identificador y el como.

O_RDONLY	(0) -> lectura
O_WRONLY	(1) -> escritura
O_RDWR	(2) -> ambos
O_CREAT	Si el fichero no existe, lo crea.

26.4 Llamadas manejo de ficheros

OPEN

Modo
rwx.rwx.rwx
00700 el propietario tiene permisos de lectura escritura y ejecución
00020 el grupo tiene permiso de escritura
00751 el propietario todo, el grupo lectura y ejecución, resto ejecución

Esta llamada devuelve un entero, el descriptor del fichero (fd) para posteriormente utilizarlo en lecturas o escrituras;
-1 en caso de error.
El descriptor de fichero devuelto es el descriptor de fichero más pequeño no abierto actualmente para el proceso.

26.4 Llamadas manejo de ficheros

CLOSE

Cierra un fichero especificado por su descriptor.

```
int close (fd)      /* cierra un fichero */
int fd;             /* descriptor del fichero */
```

devuelve -1 en caso de error, y cero en caso de exito

READ

```
int read(fd, buffer, nbytes) /* lectura de un fichero */
int fd;                  /* descriptor del fichero */
char *buffer;            /* dirección del buffer   */
unsigned nbytes;         /* numero de bytes a leer */
```

devuelve numero de bytes leídos, 0 un EOF, -1 error.

26.4 Llamadas manejo de ficheros

WRITE

int write (fd, buffer, nbytes) /* escribe en un fichero */
int fd; /* descriptor de un fichero */
char *buffer; /* dirección del buffer */
unsigned nbytes; /* numero de bytes a escribir */
devuelve el numero de bytes escritos o -1 si hay errror.

Lectura, escritura secuencial, cada vez que se lee o escribe, el pointer asociado con el fichero es incrementado n bytes.

LSEEK

Mueve el pointer del fichero a una determinada posición, utilizado para acceso directo.
long lseek(fd, offset, de_donde) /* mueve el pointer */
int fd; /* identificador del fichero */
long offset; /* desplazamiento dentro del fichero */
int de_donde; /* interpretación del offset si la posición es relativa al principio 0, o al final del fichero 2, o actual 1*/
Lseek devuelve la posición absoluta del pointer después del cambio.

26.4 Llamadas manejo de ficheros

STAT

```
int stat(nombre, p)     /* lee el estado de un fichero */
    char *nombre;       /* nombre camino */
    struct stat *p;     /* información del estado */
```

FSTAT

```
int fstat(fd, p)   /* lee el estado de un fichero */
int fd;            /* descriptor de un fichero */
struct stat *p     /* información del estado */
```

Devuelve -1 si hay error, 0 si exito

Nos da información relativa a un fichero.

Modo, directorio, tipo de fichero, tamaño, última modificación, etc.

Estas llamadas difieren solo en la forma de especificar el fichero (nombre o identificador).

26.4 Llamadas manejo de ficheros

FSTAT

El segundo parámetro es un pointer a una estructura donde va a colocarse la información. La forma de esta estructura es:

```
struct stat {
    short st_dev;          /* equipo donde el nodo esta */
    unsigned short st_ino;          /* numero del nodo */
    unsigned short st_mode;          /* palabra de modo */
    short st_nlink;              /* numero de enlaces */
    short st_uid;          /* identificador del usuario */
    short st_gid;          /* identificador del grupo */
    short st_rdev;      /* mayor/menor fichero especial */
    short st_size;              /* tamaño del fichero */
    short st_atime;              /* igual que st_mtime */
    short st_mtime;  /* fecha de la ultima modificación */
    short st_ctime;              /* hora de la última modificación del nodo-i */
};
```

26.4 Llamadas manejo de ficheros

DUP

```
int dup(fd)        /* duplica el descriptor de un fichero */
   int fd;         /* descriptor de un fichero abierto */
```

Devuelve el nuevo descriptor del fichero o -1 si hay error

Se utiliza para manejar descriptores de ficheros.

Ejemplo: Consideremos la siguiente acción a un programa.

1) Cerrar la salida standar (descriptor de fichero = 1).
2) Utilizar otro fichero como salida standar.
3) Utiliza funciones de escritura para la salida standar (printf).
4) Recuperar la situación original.

Esto se puede realizar:

- cerrando con descriptor de fichero igual a uno
- abriendo un nuevo fichero que será la nueva salida standar

pero será imposible recuperar la situación más tarde.

26.4 Llamadas manejo de ficheros

DUP

La solución correcta es:
- Ejecutar fd = dup(1)

El cual origina un nuevo descriptor para la salida standar, la cual ahora tiene dos descriptores fd y 1.
- Cerrar la salida standar close(1), (se pierde el 1).
- Abrir un fichero (open) (se recupera para éste el 1).

Para recuperar la situación original.
- Cerrar con el descriptor igual a uno.
- Ejecutar n = dup(fd) para recuperar el descriptor 1 que apunta al mismo fichero que fd.
- Cerrar con fd y volvemos a la situación original.

Esto se entiende si las asignaciones de descriptores son secuenciales, se asigna el menor descriptor disponible.

El dup tiene una variante, que permite a un descriptor, que no ha sido asignado, referenciar a un fichero abierto. **dup2 (fd,fd2);**
fd --> es el descriptor de un fichero abierto
fd2 --> el descriptor no asignado que va a referenciar al mismo fichero que fd.

26.4 Llamadas manejo de ficheros

PIPE

```
int pipe(pfd)            /* crea un pipe */
   int pfd[2];           /* descriptores de los ficheros */
```

La llamada Pipe, crea un pipe y devuelve dos enteros descriptores de fichero.

pfd[0] --> es el descriptor para leer
pfd[1] --> es el descriptor para escribir

devuelve -1 si hay error 0 si éxito.
La comunicación entre procesos se puede realizar en Linux con la utilización de pipes.

Cuando ejecutamos el comando. **cat file1 file2 | sort**
El Shell crea un pipe y hace que la salida standar del primer proceso (cat) se escriba en el pipe, y que la entrada standar del segundo proceso (sort) pueda leer de pipe.

26.4 Llamadas manejo de ficheros

PIPE

ejemplo: procedimiento que crea dos procesos, con la salida del primero, escrita sobre un pipe, y leída por el segundo.

```
#defineSTD_INPUT 0              /*fd de la entrada estandar */
#define STD_OUTPUT 1            /* fd de la salida estandar */
pipeline(proceso1, proceso2)
 char *proceso1, *proceso2;  /* punteros a nombres de programa */
{
  int fd[2];
  pipe(&fd[0]);    /* crea un pipe */
  if (fork() != 0) { /* el padre ejecuta estas sentencias */
     close(fd[0]); /* proceso uno no necesita leer de pipe */
     close(STD_OUTPUT); /*prepara para una nueva salida estandar.*/
     dup(fd[1]);  /* coloca la salida estandar a fd[1] */
     close(fd[1]); /* pipe no necesita ninguno más */
     execl(proceso1, proceso1, 0);
  } else {
     /* el proceso hijo ejecuta estas sentencias */
     close(fd[1]); /* proce. dos no necesita escribir en pipe */
     close(STD_INPUT); /*prepara para una nueva entrada estandar.*/
     dup(fd[0]);  /* coloca la entrada estandar a fd[0] */
     close(fd[0]); /* pipe no necesita ninguno más */
     execl(proceso2, proceso2, 0);
  }
}
```

26.4 Llamadas manejo de ficheros

PIPE

Escritura en un PIPE

Los datos se escriben en orden de llegada.

Si el pipe se llena (tamaño finito # 4kB), el **proceso que escribe se bloquea** hasta que datos son extraídos del pipe.

Lectura en un PIPE

Los datos se leen en el orden de llegada FIFO

Una vez un dato es leído, es sacado del pipe y no puede ser leído de nuevo.

Si el pipe esta vacio **el proceso que lee se bloquea** hasta que se introducen datos en el pipe.

Close sobre un PIPE

Libera el descriptor del pipe y puede ser asignado

Si se cierra el descriptor para escritura close(pfd[1]); actúa como un end-of-file (final de fichero) para el lector.

26.4 Llamadas manejo de ficheros

IOCTL

int ioctl(fd, comando, tbuf) /* control de un fichero especial como un terminal */

int fd; /* descriptor del fichero */
int comando; /* comando */

struct termio *tbuf; /* información del terminal */
Devuelve -1 si hay error, 0 si exito.
Tiene su aplicación mas común en terminales, se utiliza para:

- Cambiar el carácter de borrado.
- Cambiar el modo del terminal.

26.4 Llamadas manejo de ficheros

IOCTL

modo "cooked" (en POSIX modo canónico)

Para leer espera por una línea completa.

backspace --> trabaja normalmente
break --> trabaja normalmente
CRTL S --> Para la salida en el terminal
CTRL Q --> Reanuda la salida en el terminal
CTRL D --> fin de fichero
DEL --> señal de interrupción
CRTL \ --> señal quit para producir un volcado.

modo "raw" (en POSIX modo no canónico)

Todos las funciones anteriores son desabilitadas.
Los caracteres pasan directamente al programa, sin ningún proceso.
Para leer no espera por una línea completa.

26.4 Llamadas manejo de ficheros

IOCTL

La forma de esta llamada es:

ioctl (fd, operación, &sgttyb);

fd --> especifica un fichero

operación --> específica una operación

&sgttyb --> dirección a una estructura de flags

Las operaciones soportadas por son:

TIOCSETP --> asigna los valores de la estructura a los parámetros del terminal.

TIOCGETP --> Llena la estructura con los valores actuales del terminal.

La estructura está definida en el fichero sgtty.h

```
/* Estructura de datos para IOCTL llamadas TIOCGETP/TIOCSETP */
struct sgttyb {
  char sg_ispeed;  /* velocidad de entrada (no se usa) */
  char sg_ospeed;  /* velocidad de salida (no se usa) */
  char sg_erase;   /* carácter de borrado */
  char sg_kill;    /* carácter de matar */
  int  sg_flags;   /* flags de modo */
};
```

26.4 Llamadas manejo de ficheros

ACCESS

ACCESS (name,amode) - verifica los accesos de un programa a un fichero. Un programa puede ejecutarse con el UID de otro, como el caso de comandos del sistema.

RENAME

RENAME(viejo, nuevo) - cambia el nombre viejo de un fichero por nuevo.

FCNTL

FCNTL(fd, cmd, ...) - bloqueo, candados, de partes de un fichero y otras operaciones.

26.5 Llamadas manejo de directorio.

MKDIR

MKDIR(name, mode) - crea un nuevo directorio.

RMDIR

RMDIR(name) - elimina un directorio vacío.

26.5 Llamadas manejo de directorio.

LINK

Permite que varios usuarios referencien a un fichero con dos o más nombres, a menudo en diferentes directorios, el fichero no se duplica, se crea otra entrada en el directorio.
Las modificaciones afectan a todos, pues es el mismo fichero.

Ejemplo. Supongamos la siguiente situación para dos directorios.

/usr/ast			/usr/jim	
16	mail		31	bin
81	juego		70	nemo
40	test		50	fc
			38	prog1

26.5 Llamadas manejo de directorio.

LINK

después de realizar la llamada link ("/usr/jim/nemo", "/usr/ast/note")

La situación es la siguiente

/usr/ast			/usr/jim	
16	mail		31	bin
81	juego		70	nemo
40	test		50	fc
70	note		38	prog1

Todo fichero tiene un único número identificador, entrada en la tabla de inodos.
Existen dos entradas con el mismo identificador (70) que referencian al mismo fichero.

UNLINK (nombre)

Quita una entrada del directorio, si existen más, las otras permanecen.

26.5 Llamadas manejo de directorio.

MOUNT (especial, nombre, rwflag).

Realiza la fusión entre dos sistemas de ficheros (normalmente el raíz) y un sistema de ficheros de usuario, en otro dispositivo.

Ejemplo:

```
mount (/dev/fd0,  /usr/home, 0);
mount (/dev/fd0,  /fd0, 0);
```

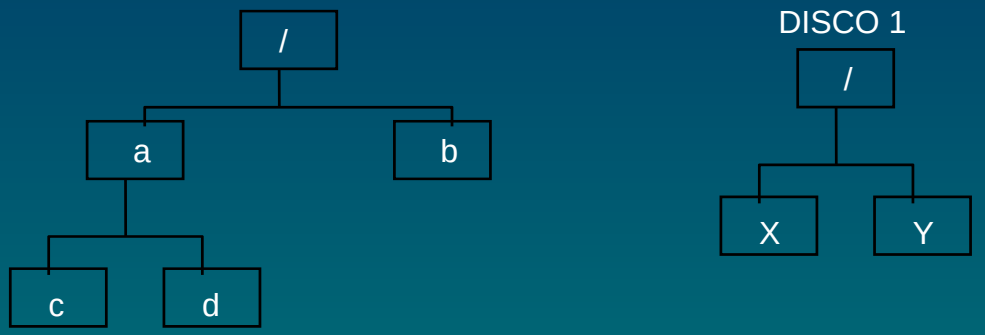
El primer parámetro es el disco donde se encuentra el sistema de ficheros a montar.
El segundo parámetro es el directorio donde se va a realizar el enlace.

Una vez que un fichero está montado forma parte del árbol general y se accede independiente del disco donde está situado.
El comando montar /etc/mount ó la llamada al sistema MOUNT, permiten enganchar un sistema de ficheros en disco con el árbol principal, situado en memoria RAM.

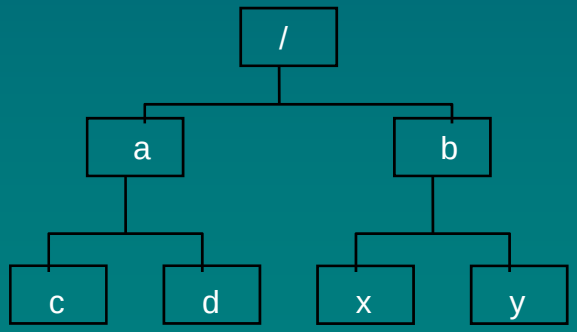
26.5 Llamadas manejo de directorio.

MOUNT

Dada la situación de la figura.



Si ejecutamos el comando **/bin/mount /dev/fd0 /b** la situación final es:



26.5 Llamadas manejo de directorio.

UMOUNT (/dev/fd0)

Si se desea quitar el disco,
hay que realizar la operación inversa desmontar el disco
o bien con la llamada al sistema UMOUNT

ó con el comando:

desmontar	**disco 1**
/bin/umount	**/dev/fd0**

UMOUNT (/dev/fd0)
Desmonta un sistema de ficheros.

26.5 Llamadas manejo de directorio.

SYNC ()

El sistema mantiene los datos usado más recientemente del disco, en memoria principal. (memoria cache, buffer).
Si se modifica un dato de esta memoria, y antes que se guarde en el disco, el sistema cae, se puede producir un error, para prevenir esto; SYNC escribe bloques, de datos que han sido modificados, de la memoria cache a disco.
Cuando el sistema se arranca, se ejecuta el programa update en modo tanda, y este ejecuta un SYNC cada 30 segundos.

CHDIR (nombre de directorio).
Cambia el directorio de trabajo,después de la llamada
chdir ("/usr/ast/test");
un open sobre el fichero xyz, lo abre en /usr/ast/test/xyz

CHROOT ("/usr")
Cambia el directorio raíz "/", solo el super usuarios pueden realizar esta llamada.

26.6 Llamadas para protección

CHMOD (nombre, modo)

El sistema proporciona 11 bits de modo, para protección de un fichero. La llamada CHMOD, permite cambiar estos bits de modo para un fichero.

Los nueve primeros bits, son para el propietario, el grupo y el resto de usuarios (rwx).

Ejemplo: **chmod ("file", 0644);**

Coloca el fichero file con acceso de solo lectura para cualquiera, excepto para el propietario.

Los otros dos bits son:

bit (10) 02000 SETGID (pone identificador de grupo)

bit (11) 04000 SETUID (pone identificador de usuario)

Cuando un usuario, ejecuta un programa con el bit **SETUID** puesto a uno, el identificador del usuario, se cambia por el identificador del dueño del fichero, y este cambio dura la vida del proceso.

Esto se utiliza, para permitir que un usuario pueda ejecutar procesos del super usuario como por ejemplo crear un directorio. Si a **mkdir** se le coloca el modo 04755, usuarios normales pueden ejecutarlo.

26.6 Llamadas para protección

SETGID cuando está a uno, el usuario del fichero, cambia su identificador de grupo, por el del propietario del fichero.

GETUID () GETGID ()
Dan al proceso que hace esta llamada, el identificado real y efectivo del usuario, y grupo respectivamente.
Se necesitan cuatro rutinas de librería para obtener esta información.
getuid, getgid, geteuid, getegid
Las dos primeras cogen el real uid/gid y las dos últimas el efectivo uid/gid.

SETUID(uid) SETGID(gid)
Estas llamadas permiten al super usuario, cambiar el identificador real y efectivo de un usuario y grupo.

CHOWN (nombre, propietario, grupo)
Con esta llamada, el super usuario puede cambiar el propietario de un fichero.

26.6 Llamadas para protección

UMASK (modo máscara)

Pone una máscara interna de bits, en el sistema, para enmascarar los bits de modo, cuando se crea un fichero.

Ejemplo, la llamada **umask (022)**
Hace que una llamada posterior para crear un fichero.
creat ("file", 0777)
Su modo 0777 sea enmascarado con 022 y por tanto cambiado a 0755.

```
          0777  =  1 1 1   1 1 1   1 1 1
mácara:   0022  =  1 1 1   1 0 1   1 0 1
          and      1 1 1   1 0 1   1 0 1  =  755
```

Los bits de máscara son heredados por los procesos hijos.
Si el Shell hace umask justo después de login, ninguno de los procesos de usuario en esa sesión crearan ficheros que accidentalmente pueden ser escritos por otros usuarios.

26.6 Llamadas para protección

ACCESS

Esta llamada nos dice si un usuario tiene permiso de acceso para un fichero.

Si un programa del super usuario, tiene el SETUID a uno, cualquier usuario que la ejecute tomara el identificador efectivo del super usuario y podrá ejecutar cualquier programa.

La llamada access nos dice si un identificador real tiene permiso para acceder a un fichero.

El parámetro modo es 4 para lectura, 2 para escritura y 1 para ejecutar, se permiten combinaciones por ejemplo con modo 6, si la llamada no devuelve un cero, si se permite acceso de lectura y escritura y 1 en otro caso.

Con modo cero, se testea si el fichero existe.

26.7 Llamadas para manejo de reloj

TIME (&segundos)
Nos devuelve los segundos transcurridos desde el uno de enero de 1970.

STIME (segundos)
El super usuario pone el tiempo transcurrido, desde 1 de Enero de 1970, para posteriores lecturas.

UTIME (fichero, fecha)
Permite el usuario de un fichero, escribir en el campo "última fecha de acceso del fichero", en el directorio.
Algunos programas exigen que todos sus ficheros tengan la misma fecha de modificación.

TIMES (buffer)
Devuelve información a un proceso como:
tiempo de CPU gastado por el proceso
tiempo de CPU gastado por el sistema
tiempo de CPU de los procesos hijos.