Guía Linux de Programación (GULP)

Sven Goldt
Sven van der Meer
Scott Burkett
Matt Welsh

Versión 0.4
Marzo 1995

[0]...Nuestro objetivo permanente: mejorar nuestro conocimiento de C, explorar extraños comandos Unix y to boldly code where no one has man page 4.

Índice General

- Copyright
  La Guía Linux de Programación es © 1994, 1995 de Sven Goldt
  Sven Goldt, Sachsendamm 47b, 10829 Berlín, Alemania
  $< goldt@math.tu - berlin.de >$.
  El capítulo 8 es © 1994, 1995 de Sven van der Meer $< vdmeer@cs.tu - berlin.de >$.
  El capítulo 6 es © 1995 de Scott Burkett $< scottb@IntNet.net >$.
  El capítulo 10 es © 1994, 1995 de Matt Welsh $< mdw@cs.cornell.edu >$.
  Tenemos que dar especialmente las gracias a John D. Harper $< jharper@uiuc.edu >$ por revisar en profundidad esta guía.
  Se concede permiso para reproducir este documento, en todo o en parte, bajo las siguientes condiciones:

  1. Esta nota de Copyright debe incluirse sin modificaciones.
  2. Comparta con los autores cualquier ganancia que obtenga.
  3. Los autores no se hacen responsables de cualquier daño producido en aplicación de los contenidos de este libro.

- Copyright (nota original)
  The Linux Programmer's Guide is © 1994, 1995 by Sven Goldt
  Sven Goldt, Sachsendamm 47b, 10829 Berlin, Germany
  $< goldt@math.tu - berlin.de >$.
  Chapter 8 is © 1994, 1995 by Sven van der Meer $< vdmeer@cs.tu - berlin.de >$.
  Chapter 6 is © 1995 Scott Burkett $< scottb@IntNet.net >$.
  Chapter 10 is © 1994, 1995 Matt Welsh $< mdw@cs.cornell.edu >$.
  Special thanks goes to John D. Harper $< jharper@uiuc.edu >$ for proofreading this guide.
  Permission to reproduce this document in whole or in part is subject to the following conditions:

  1. The copyright notice remains intact and is included.
  2. If you make money with it the authors want a share.
  3. The authors are not responsible for any harm that might arise by the use of it.

- Notas sobre la versión castellana
  Esta guía, como cuarto trabajo importante del Proyecto LuCAS, obedece a la demanda de guías de programación para Unix/Linux que venimos observando desde tiempos recientes. Sin embargo, lamentamos que nuestra traducción sea tan incompleta como la versión original en Inglés: ciertamente nos gustaría completarla, sin embargo no hemos podido recibir los permisos necesarios para ello de algunos de sus autores originales, al estar actualmente ilocalizables. El proyecto LuCAS agradece el trabajo de traducción realizado inicialmente por Pedro Pablo Fábrega[1], que abarca buena parte del libro. Además, agradecemos la colaboración prestada por

[1] Pedro Pablo está disponible en pfabrega@arrakis.es

Ignacio Arenaza, César Ballardini y Luis Francisco González[2], quienes se han ocupado de la traducción del resto del libro.
Nota: Versión de la traducción: 0.11 *alpha*
Juan José Amor[3], Mayo de 1998.

- Prólogo
  Esta guía está lejos de completarse.
  La primera edición fue la versión 0.1, de septiembre de 1994. Se basó en las llamadas al sistema debido a la escased de información al respecto. Está previsto completarla con la descripción de las funciones de librería y cambios importantes en el núcleo, así como incursiones en áreas como redes, sonido, gráficos y entrada/salida asíncrona. Asímismo, se incluirán en un futuro apuntes sobre cómo construir librerías dinámicas y acerca de interesantes herramientas para el programador.
  Esta guía solo será un éxito gracias a la ayuda en forma de información o de envío de nuevos capítulos.

- Introducción
  En cierta ocasión me dispuse a instalar Linux en mi PC para aprender más acerca de administración del sistema. Intenté instalar un servidor de SLIP pero no trabajé con *mgetty* ni con el *shadow*. Tuve que parchear el *sliplogin* y funcionó hasta las nuevas versiones de Linux 1.1. Nadie me explicó qué había pasado. No había documentación acerca de los cambios desde el núcleo 0.99 salvo los resúmenes que hacía Russ Nelson, si bien éstos no me ayudaban demasiado a resolver mis problemas.

  La Guía Linux de Programación pretende servir para lo que su nombre implica— para ayudar al programador de Linux a entender las peculiaridades de este sistema operativo. También deberá ser útil para transportar programas de otros sistemas operativos al Linux. Por lo tanto, esta guía debe describir las llamadas al sistema y los cambios importantes del núcleo que puedan afectar a antiguos programas tales como aplicaciones de E/S serie o de red.

Sven Goldt Guía Linux de Programación

[2]Sus direcciones de correo respectivas son: inaki.arenaza@jet.es, cballard@santafe.com.ar y luisgh@cogs.susx.ac.uk
[3]Como siempre, en jjamor@ls.fi.upm.es

Capítulo 1

El sistema operativo Linux

En marzo de 1991 Linus Benedict Torvalds compró un sistema Multitarea Minix para su AT. Lo usó para desarrollar su propio sistema multitarea que llamó Linux. En el mes septiembre de 1991 liberó el primer prototipo por e-mail a algunos otros usuarios de Minix en Internet: así comenzó el proyecto Linux. Muchos programadores desde ese punto han apoyado Linux. Han agregado controladores de dispositivos, desarrollado aplicaciones, según las normas POSIX. Hoy Linux es muy potente, pero lo mejor es que es gratuito. Se están realizando trabajos para transportar Linux a otras plataformas.

Capítulo 2

El núcleo de Linux

La base de Linux es el núcleo. Podría reemplazar todas las librerías, pero mientras quede el núcleo, estará todavía Linux. El núcleo incorpora controladores de dispositivos, manejo de la memoria, manejo de procesos y manejo de comunicaciones. Los gurús del núcleo siguen las pautas POSIX que hacen la programación a veces más fácil y a veces más difícil. Si su programa se comporta de forma diferente en un nuevo núcleo de Linux, puede ser porque se hayan implantado nuevas líneas marcadas por POSIX. Para más información de la programación sobre el núcleo de Linux, lea el documento Linux Kernel Hacker's Guide.

Capítulo 3

El paquete libc de Linux

> libc: ISO 8859.1, $<linux/param.h>$, funciones YP, funciones crypt, algunas rutinas shadow básicas (por omisión no incluidas),... rutinas viejas por compatibilidad en libcompat (por omisión no activas), mensajes del error en inglés, francés o alemán, rutinas de gestión de la pantalla compatibles bsd 4.4lite en libcurses, rutinas compatibles bsd en libbsd, rutinas de la manipulación de la pantalla en libtermcap, rutinas del manejo del base de datos en libdbm, rutinas matemáticas en libm, entradas para ejecutar programas en crt0.o???, información del sexo del byte en libieee??? (¿podía alguien dar información en lugar de reírse?), espacio de perfiles de usuario, en libgmon. Me gustaría que alguno de los desarrolladores de la librería libc de Linux escribiera este capítulo. Todo lo que puedo decir ahora es que va ha haber un cambio del formato de ejecutables a.out a elf (formato ejecutable y enlazable) que también significa un cambio en la construcción de bibliotecas compartidas. Normalmente se soportan ambos formatos, a.out y elf

La mayoría de los elementos del paquete libc de Linux están bajo la Licencia Pública GNU, aunque algunos están bajo una excepción especial de derechos de copia como crt0.o. Para distribuciones comerciales binarias esto significa una restricción que prohibe el enlace estático de ejecutables. El enlace dinámico de ejecutables son de nuevo una excepción especial y Richard Stallman del FSF comentó:
*[...] Pero me parece que debemos permitir de forma ambigua la distribución de ejecutables enlazados dinámicamente *sin* ir acompañados de la librerías bibliotecas, con tal de que los ficheros objeto que forman el ejecutable estén sin restricción según la sección 5 [...] Por tanto tomaré la decisión de permitirlo. La actualización del LGPL tendrá que esperar hasta que tenga tiempo para hacer y comprobar una versión nueva.*

Sven Goldt Guía Linux de Programación

Capítulo 4

Llamadas al sistema

Una llamada al sistema es normalmente una demanda al sistema operativo (nucleo) para que haga una operación de hardware/sistema específica o privilegiada. Por ejemplo, en Linux-1.2, se han definido 140 llamadas al sistema. Las llamadas al sistema como close() se implementan en la libc de Linux. Esta aplicación a menudo implica la llamada a una macro que puede llamar a syscall(). Los parámetros pasados a syscall() son el número de la llamada al sistema seguida por el argumento necesario. Los números de llamadas al sistema se pueden encontrar en $< linux/unistd.h >$ mientras que $< sys/syscall.h >$ actualiza con una nueva libc. Si aparecen nuevas llamadas que no tienen una referencia en libc aun, puede usar syscall(). Como ejemplo, puede cerrar un fichero usando syscall() de la siguiente forma (no aconsejable):

```
#include <syscall.h>

extern int syscall(int, ...);

int my_close(int filedescriptor)
{
   return syscall(SYS_close, filedescriptor);
}
```

En la arquitectura i386, las llamadas al sistema están limitadas a 5 argumentos además del número de llamada al sistema debido al número de registros del procesador. Si usa Linux en otra arquitectura puede comprobar el contenido de $< asm/unistd.h >$ para las macros _syscall, para ver cuántos argumentos admite su hardware o cuantos escogieron los desarrolladores. Estas macros _syscall se pueden usar en lugar de syscall(), pero esto no se recomienda ya que esa macro se expande a una función que ya puede existir en una biblioteca. Por consiguiente, sólo los desarrolladores del núcleo deberían jugar a con las macros _syscall. Como demostración, aquí tenemos el ejemplo de close() usando una macro _syscall.

```
#include <linux/unistd.h>

_syscall1(int, close, int, filedescriptor);
```

La macro _syscall1 expande la función close(). Así tenemos close() dos veces, una vez en libc y otra vez en nuestro programa. El valor devuelto por syscall() o un una macro _syscall es -1 si la llamada al sistema falló y 0 en caso de éxito. Déle un vistazo a la variable global errno para comprobar que ha ocurrido si la llamada al sistama falló.

Las siguiente llamadas al sistema están disponibles en BSD y SYS V pero no están disponibles en Linux:
audit(), auditon(), auditsvc(), fchroot(), getauid(), getdents(), getmsg(), mincore(), poll(), putmsg(), setaudit(), setauid().

Sven Goldt Guía Linux de Programación

Capítulo 5

Una llamada multiuso: "ioctl"

ioctl representa el control de entrada/salida y se usa para manipular un dispositivo de carácter mediante un descriptor de fichero. El formato de ioctl es:
ioctl(unsigned int fd, unsigned int request, unsigned long argument).
El valor devuelto es -1 si ocurrió un error y un valor mayor o igual que 0 si la petición tuvo éxito, como cualquier otra llamadas del sistema. El núcleo distingue entre ficheros especiales y regulares. Los ficheros especiales se encuentran principalmente en /dev y /proc. Difieren de los ficheros regulares en que esconden una interface a un controlador y no un fichero real (regular) que contiene texto o datos binarios. Esta es la filosofía UNIX y permite usar operaciones normales de lectura/escritura en cada fichero. Pero si necesita hacer algo más con un fichero especial o un fichero regular que puede hacer él con... sí, ioctl. Usted necesitará con más frecuencia ioctl para ficheros especiales que para ficheros regulares, pero es posible usar ioctl en ficheros regulares también.

Capítulo 6

Comunicación entre procesos en Linux

B. Scott Burkett, `scottb@intnet.net` v1.0, 29 de Marzo de 1995

6.1 Introducción

Los medios IPC (Inter-process communication) de Linux proporcionan un método para que múltiples procesos se comuniquen unos con otros. Hay varios métodos de IPC disponibles para los programadores Linux en C:

- Pipes UNIX Half-duplex
- FIFOs (pipes con nombre)
- Colas de mensajes estilo SYSV
- Semáforos estilo SYSV
- Segmentos de memoria compartida estilo SYSV
- Sockets (estilo Berkeley) (no contemplado por ahora)
- Pipes Full-duplex (pipes STREAMS) (no contemplado por ahora)

Estos medios, cuando se usan de forma efectiva, proporciona una base sólida para el desarrollo de cliente/servidor en cualquier sistema UNIX (incluido Linux).

6.2 Pipes UNIX Semi-dúplex

6.2.1 Conceptos básicos

Simplemente, una tubería (*pipe*) es un método de conexión de que une la *salida estándar* de un proceso a la *entrada estándar* de otro. Las tuberías son la mayor de las herramientas de IPC, han estado presentes desde los primeros orígenes del sistema operativo UNIX. Proporcionan un método de comunicaciones en un sentido (unidirecional, semi-duplex) entre procesos.

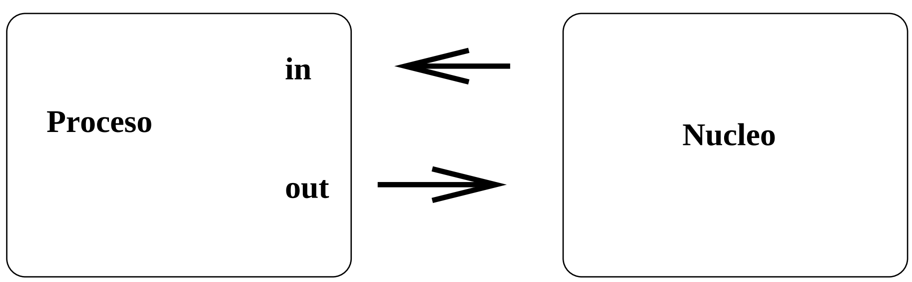
Este mecanismo es ampliamente usado, incluso en la línea de comandos UNIX (en la shell):

```
ls | sort | lp
```

Lo anterior es un ejemplo de 'pipeline', donde se toma la salida de un comando `ls` como entrada de un comando `sort`, quien a su vez entrega su salida a la entrada de `lp`. Los datos corren por la tubería semi-duplex, de viajando (virtualmente) de izquierda a derecha por la tubería.

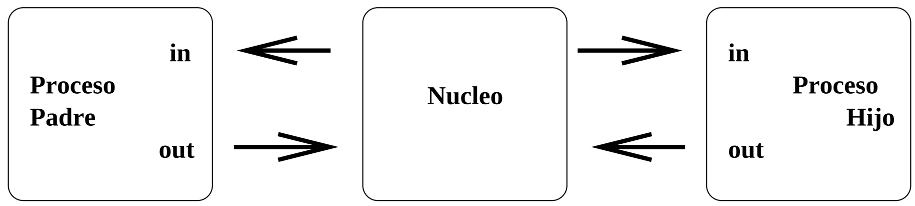
Aunque la mayor parte de nosotros usamos las tuberías casi religiosamente en las programaciones de scripts de shell, casi nunca nos paramos a pensar en lo que tiene lugar a nivel del núcleo.

Cuando un proceso crea una tubería, el núcleo instala dos descriptores de ficheros para que los use la tubería. Un descriptor se usa para permitir un camino de entrada a la tubería (write), mientras que la otra se usa para obtener los datos de la tubería (read). A estas alturas, la tubería tiene un pequeño uso práctico, ya que la creación del proceso sólo usa la tubería para comunicarse consigo mismo. Considere esta representación de un proceso y del núcleo después de que se haya creado una tubería:

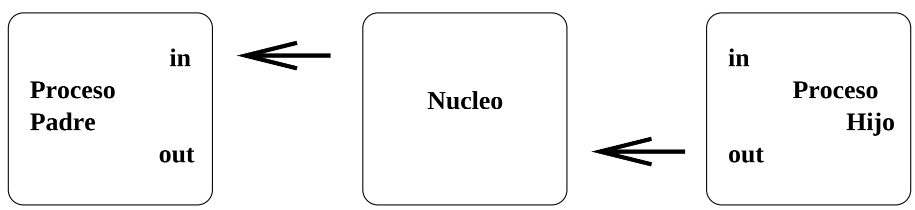


Del diagrama anterior, es fácil ver cómo se conectan los descriptores. Si el proceso envía datos por la tubería (fd0), tiene la habilidad obtener (leer) esa información de fd1. Sin embargo, hay un objetivo más amplio sobre el esquema anterior. Mientras una tubería conecta inicialmente un proceso a sí mismo, los datos que viajan por la tubería se mueven por el núcleo. Bajo Linux en particular, las tuberías se representan realmente de forma interna con un inodo válido. Por supuesto, este inodo reside dentro del núcleo mismo, y no dentro de los límites de cualquier sistema de archivos físico. Este punto particular nos abrirá algunas puertas de E/S bastante prácticas, como veremos un poco más adelante.

A estas alturas, la tubería es bastante inútil. Después de todo ¿por qué el problema de crear una cañería si estamos sólo hablando con nosotros mismos? Ahora, el proceso de creación bifurca un proceso hijo. Como un proceso hijo hereda cualquier descriptor de fichero abierto del padre, ahora tenemos la base por comunicación multiprocesos (entre padre e hijo). Considere éste versión actualizada de de nuestro esquema simple:



Arriba, vemos que ambos procesos ahora tienen acceso al descriptor del fichero que constituye la tubería. Está en esa fase, que se debe tomar una decisión crítica. ¿En que dirección queremos que viajen los datos? ¿El proceso hijo envía información al padre, o viceversa? Los dos procesos mutuamente están de acuerdo en esta emisión, y procede a“cerrar” el extremo de la cañería que no le interesa. Por motivos discusión, digamos que el hijo ejecuta unos procesos, y devuelve información por la tubería al padre. Nuestro esquema ya revisado aparecería como:



¡Ahora la construcción de la tubería está completa! Lo único que queda por hacer es usar la tubería. Para a acceder a una tubería directamente, podemos usar la misma llamada al sistema que se usa para un fichero I/O de bajo nivel. (las tuberías están representadas internamente como un inodo válido).

Para enviarle datos a la tubería, usamos la llamada al sistema write(), y para recuperar datos de la tubería, usamos la llamada al sistema read(). ¡Recuerde las llamadas del sistema a los ficheros I/O de bajo-nivel se hacen usando descriptores de fichero! Sin embargo, tenga presente que ciertas llamadas al sistema, como por ejemplo lseek(), no trabaja con descriptores a tuberías.

6.2.2 Creación de tuberías en C

Crear “tuberías” con el lenguaje de programación C puede ser un poco más complejo que en un ejemplo de shell. Para crear una tubería simple con C, hacemos uso de la llamada al sistema pipe(). Toma un argumento solo, que es una tabla de dos enteros, y si tiene éxito, la tabla contendrá dos nuevos descriptores de ficheros para ser usados por la tubería. Después de crear una tubería, el proceso típicamente desdobla a un proceso nuevo (recuerde que el hijo hereda los descriptores del fichero).

```
LLAMADA AL SISTEMA: pipe();

PROTOTIPO: int pipe( int fd[2] );
```

```
RETORNA: 0  si éxito
        -1 si error: errno = EMFILE (no quedan descriptores libres)
                             EMFILE (tabla de ficheros del sistema llena)
                             EFAULT (el vector fd no es válido)

  NOTAS: fd[0] es para leer, fd[1] es para escribir
```

El primer del vector fd (elemento 0) está fijado y abierto para lectura, mientras el segundo entero (elemento 1) está fijado y abierto para escritura. Visualmente hablando, la salida de fd1 se vuelve la entrada para fd0. Una vez más, todo datos que se mueven por la tubería los hacen por el núcleo.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

main()
{
        int     fd[2];

        pipe(fd);
        .
        .
}
```

Recuerde que un nombre de vector en C es un puntero a su primer miembro. Es decir, fd es equivalente a &fd0. Una vez hemos establecido la tubería, entonces desdoblamos (fork) nuestro nuevo proceso hijo:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
        .
        .
}
```

Si el padre quiere recibir datos del hijo, debe cerrar fd1, y el hijo debe cerrar fd0. Si el padre quiere enviarle datos al hijo, debe cerrar fd0, y el hijo debe cerrar fd1. Como los descriptores se comparten entre el padre y hijo, siempre debemos estar seguros cerrar el extremo de cañería que no nos interesa. Como nota técnica, nunca se devolverá EOF si los extremos innecesarios de la tubería no son explicitamente cerrados.

```
        #include <stdio.h>
        #include <unistd.h>
        #include <sys/types.h>

        main()
        {
                int     fd[2];
                pid_t   childpid;

                pipe(fd);

                if((childpid = fork()) == -1)
                {
                        perror("fork");
                        exit(1);
                }

                if(childpid == 0)
                {
                        /* El hijo cierra el descriptor de entrada */
                        close(fd[0]);
                }
                else
                {
                        /* El padre cierra el descriptor de salida */
                        close(fd[1]);
                }
                .
                .
        }
```

Como se mencionó previamente, una vez se ha establecido la tubería, los descriptores de fichero se tratan como descriptores a ficheros normales.

```
/*****************************************************************************
 Parte de la "Guia Linux de Programacion - Capitulo 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULO: pipe.c
```

```
 ****************************************************************************/

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hola a todos!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Cierre del descriptor de entrada en el hijo */
                close(fd[0]);

                /* Enviar el saludo via descriptor de salida */
                write(fd[1], string, strlen(string));
                exit(0);
        }
        else
        {
                /* Cierre del descriptor de salida en el padre  */
                close(fd[1]);

                /* Leer algo de la tuberia... el saludo! */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
}
```

A menudo, los descriptores del hijo son duplicados en la entrada o salida estándares. El hijo puede entonces hacer exec() con otro programa, que hereda los stream estándar. Observe la llamada al sistema dup():

```
LLAMADA AL SISTEMA: dup();

PROTOTIPO: int dup( int oldfd );
  RETORNA: nuevo descriptor si hay éxito
           -1 si error: errno = EBADF (oldfd no es un descriptor valido)
                                EBADF (newfd se sale del rango)
                                EMFILE (Hay demasiados descriptores en el proceso abiertos)

NOTAS: ¡el antiguo descriptor no se cierra! Asi podemos intercambiarlos
```

Aunque el descriptor viejo y el recién creado se puede intercambiar, normalmente cerraremos primero uno de los stream estándar. La llamada al sistema dup() usa el número descriptor más bajo no utilizado para el nuevo descriptor.

Considere lo siguiente:

```
      .
      .
      childpid = fork();

      if(childpid == 0)
      {
              /* Cerrar la entrada estandar en el hijo */
              close(0);

              /* Duplicar sobre esta la salida de la tuberia */
              dup(fd[0]);
              execlp("sort", "sort", NULL);

      }
```

Como el descriptor de fichero 0 (stdin) se cerró, la llamada a dup() duplicó el descriptor de la entrada de la tubería (fd0) hacia su entrada estándar. Entonces hacemos una llamada a execlp() recubrir el segmento de texto (código) del hijo con el del programa. ¡Desde no hace mucho los programas exec heredan los stream estándares de sus orígenes, realmente hereda el lado de la entrada de la tubería como su entrada estándar! Ahora, cualquier cosa que el padre original procesa lo envía a la tubería, va en la facilidad de la clase.

Hay otra llamada al sistema, dup2 (), que se puede usar también. Esta llamada particular tiene su origen con la Versión 7 de UNIX, se realizó por una versión de BSD y ahora es requerida por el estándar POSIX.

```
LLAMADA AL SISTEMA: dup2();

PROTOTIPO: int dup2( int oldfd, int newfd );
  RETORNA: nuevo descriptor si hay éxito
           -1 si error: errno = EBADF (oldfd no es descriptor válido)
                                EBADF (newfd está fuera de rango)
```

```
                                        EMFILE (demasiados descriptores abiertos)

  NOTAS: ¡el descriptor antiguo es cerrado con dup2()!
```

Con esta particular llamada, tenemos la operación de cerrado, y la duplicación del descriptor actual, relacionado con una llamada al sistema. Además, se garantiza el ser atómica, que esencialmente significa que nunca se interrumpirá por la llegada de una señal. Toda la operación transcurrirá antes de devolverle el control al núcleo para despachar la señal. Con la llamada al sistema dup() original, los programadores tenían que ejecutar un close() antes de llamarla. Esto resultaba de dos llamadas del sistema, con un grado pequeño de vulnerabilidad en el breve tiempo que transcurre entre ellas. Si llega una señal durante ese tiempo, la duplicación del descriptor fallaría. Por supuesto, dup2 () resuelve este problema para nosotros.

Considere:

```
        .
        .
        childpid = fork();

        if(childpid == 0)
        {
                /* Cerrar entrada estandar, duplicando a esta la
                   salida de datos de la tuberia */
                dup2(0, fd[0]);
                execlp("sort", "sort", NULL);
                .
                .
        }
```

6.2.3 Tuberías, la forma fácil de hacerlo

Si de todo lo visto anteriormente parece enredada la forma de crear y utilizar tuberías, hay una alternativa:

```
  FUNCIÓN DE LIBRERÍA: popen();

  PROTOTIPO: FILE *popen ( char *comando, char *tipo);
    RETORNA: si hay éxito, nuevo "stream" de fichero
             si no hay éxito, NULL (por fallo en llamada pipe() o fork() ).

  NOTAS: crea una tubería, y realiza las llamadas fork/exec
         según el "comando" pasado como argumento.
```

Esta función estándar de la biblioteca crea una tubería semi-duplex llamando a pipe() internamente. Entonces adesdobla un proceso hijo, abre una shell Bourne y ejecuta el argumento “command” en la shell. La dirección del flujo de datos se determina por el segundo argumento, "type". Puede ser "r" o "w", para "read" o "write". ¡No pueden ser ambos!. Bajo Linux la tubería se abrirá según el modo especificado por el primer carácter del argumento "type". Así, si trata de pasar "rw", sólo lo abre en modo "read".

Mientras esta función de la biblioteca ejecuta realmente parte del trabajo sucio por usted, hay un inconveniente substancial. Pierde parte del control que tenía con el uso de la llamada al sistema pipe(), y la manipulación de fork/exec por usted mismo. Sin embargo, como la shell de Bourne se usa directamente, la expansión de metacaracteres de la shell (incluso plantillas) está permitida dentro del argumento "comando".

Las tuberías que se crean con popen() se debe cerrar con pclose(). Por ahora, probablemente se habrá dado cuenta de que popen/pclose comparten un parecido llamativo con las funciones I/O stream de fichero normal fopen() y fclose().

```
  FUNCIÓN DE LIBRERÍA: pclose();

  PROTOTIPO: int pclose( FILE *stream );
    RETORNA: el código de retorno de la llamada wait4()
             -1 si el "stream" pasado no es válido, o la llamada wait4() falla

  NOTAS: espera a que el proceso que escribe en la tuberia termine,
  y luego cierra el "stream".
```

La función pclose() ejecuta un wait4() en el proceso desdoblado por popen(). Cuando vuelve, destruye la tubería y el stream de fichero de salida. Una vez más, es sinónimo de la función fclose() para ficheros E/S normales de stream.

Considere este ejemplo, que abre una tubería al comando sort, y ordena un array de cadena de caracteres.:

```
/*****************************************************************************
 Parte de la "Guia Linux de Programacion - Capitulo 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULO: popen1.c
 *****************************************************************************/

#include <stdio.h>

#define MAXSTRS 5

int main(void)
{
        int  cntr;
        FILE *pipe_fp;
        char *strings[MAXSTRS] = { "eco", "bravo", "alpha",
```

```
                                   "charlie", "delta"};

        /* Crea una tuberia de un sentido llamando a popen() */
        if (( pipe_fp = popen("sort", "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }

        /* Bucle de proceso */
        for(cntr=0; cntr<MAXSTRS; cntr++) {
                fputs(strings[cntr], pipe_fp);
                fputc('\n', pipe_fp);
        }

        /* Cierra la tuberia */
        pclose(pipe_fp);

        return(0);
}
```

Como `popen()` usa la shell para hacer su enlace, ¡todas las expansiones de caracteres y metacaracteres de la shell están disponibles para su uso! Además, técnicas más avanzadas tales como redirección, e incluso la salida por tubería se puede utilizar con `popen()`. Considere el siguiente ejemplo:

```
        popen("ls ~scottb", "r");
        popen("sort > /tmp/foo", "w");
        popen("sort | uniq | more", "w");
```

Considere este pequeño programa como otro ejemplo de popen(), que abre dos tuberías (una a la orden ls, el otro a sort):

```
/*****************************************************************************
 Parte de la "Guia Linux de Programacion - Capitulo 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULO: popen2.c
 *****************************************************************************/

#include <stdio.h>

int main(void)
{
        FILE *pipein_fp, *pipeout_fp;
        char readbuf[80];
```

```
        /* Crea una tuberia de un sentido llamando a popen() */
        if (( pipein_fp = popen("ls", "r")) == NULL)
        {
                perror("popen");
                exit(1);
        }

        /* Crea una tuberia de un sentido llamando a popen() */
        if (( pipeout_fp = popen("sort", "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }

        /* Bucle de proceso */
        while(fgets(readbuf, 80, pipein_fp))
                fputs(readbuf, pipeout_fp);

        /* Cierre de las tuberias */
        pclose(pipein_fp);
        pclose(pipeout_fp);

        return(0);
}
```

Para nuestra demostración final de popen(), creamos un programa genérico que abre una tubería entre una orden pasada y un nombre de fichero:

```
/*****************************************************************************
 Parte de la "Guia Linux de Programacion - Capitulo 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULO: popen3.c
 *****************************************************************************/

#include <stdio.h>

int main(int argc, char *argv[])
{
        FILE *pipe_fp, *infile;
        char readbuf[80];

        if( argc != 3) {
                fprintf(stderr, "USO:  popen3 [comando] [archivo]\n");
                exit(1);
```

```
        }

        /* Abrir el fichero de entrada */
        if (( infile = fopen(argv[2], "rt")) == NULL)
        {
                perror("fopen");
                exit(1);
        }

        /* Crear una tuberia de un sentido llamando a popen() */
        if (( pipe_fp = popen(argv[1], "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }

        /* Bucle de proceso */
        do {
                fgets(readbuf, 80, infile);
                if(feof(infile)) break;

                fputs(readbuf, pipe_fp);
        } while(!feof(infile));

        fclose(infile);
        pclose(pipe_fp);

        return(0);
}
```

Pruebe este programa, con las llamadas siguientes:

```
popen3 sort popen3.c
popen3 cat popen3.c
popen3 more popen3.c
popen3 cat popen3.c | grep main
```

6.2.4 Operaciones atómicas con tuberías

Para que una operación se considere "atómica", no se debe interrumpir de ninguna manera. Todo su funcionamiento ocurre de una vez. La norma POSIX indica en `/usr/include/posix1_lim.h` que el tamaño máximo del buffer para una operación atómica en una tubería es:

```
#define _POSIX_PIPE_BUF         512
```

Hasta 512 bytes se pueden escribir o recuperar de una tubería atómicamente. Cualquier cosa que sobrepase este límite se partirá. Bajo Linux sin embargo, se define el límite atómico operacional en "linux/limits.h" como:

```
#define PIPE_BUF        4096
```

Como puede ver, Linux adapta el número mínimo de bytes requerido por POSIX, y se le pueden agregar bastantes. La atomicidad del funcionamiento de tubería se vuelve importante cuando implica más de un proceso (FIFOS). Por ejemplo, si el número de bytes escritos en una tubería excede el límite atómico para una simple operación, y procesos múltiples están escribiendo en la tubería, los datos serán "intercalados" o "chunked". En otras palabras, un proceso insertaría datos en la tubería entre la escritura de otro.

6.2.5 Notas acerca de las tuberías semi-dúplex:

- Se pueden crear tuberías de dos direcciones abriendo dos tuberías, y reasignado los descriptores de fichero al proceso hijo.
- La llamada a pipe() debe hacerse ANTES de la llamada a fork(), o los hijos no heredarán los descriptores (igual que en popen()).
- Con tuberías semi -duplex, cualquier proceso conectado debe compartir el ancestro indicado. Como la tubería reside en el núcleo, cualquier proceso que no sea ancestro del creador de la tubería no tiene forma de direccionarlo. Este no es el caso de las tuberías con nombre (FIFOS).

6.3 Tuberías con Nombre (FIFO - First In First Out)

6.3.1 Conceptos básicos

Una tubería con nombre funciona como una tubería normal, pero tiene algunas diferencias notables.

- Las tuberías con nombre existen en el sistema de archivos como un archivo de dispositivo especial.
- Los procesos de diferentes padres pueden compartir datos mediante una tubería con nombre.
- Cuando se han realizados todas las I/O por procesos compartidos, la tubería con nombre permanece en el sistema de archivos para un uso posterior.

6.3.2 Creación de una FIFO

Hay varias formas de crear una tubería con nombre. Las dos primeras se pueden hacer directamente de la shell.

```
mknod MIFIFO p
mkfifo a=rw MIFIFO
```

Los dos comandos anteriores realizan operaciones idénticas, con una excepción. El comando mkfifo proporciona una posibilidad de alterar los permisos del fichero FIFO directamente tras la creación. Con mknod será necesaria una llamada al comando chmod.

Los ficheros FIFO se pueden identificar rápidamente en un archivo físico por el indicador "p" que aparece en la lista del directorio.

```
$ ls -l MIFIFO
prw-r--r--   1 root     root            0 Dec 14 22:15 MIFIFO|
```

También hay que observar que la barra vertical ("símbolo pipe") está situada inmediatamente detrás del nombre de fichero. Otra gran razón para usar Linux ¿eh?

Para crear un FIFO en C, podemos hacer uso de la llamada del sistema mknod():

```
FUNCIÓN DE LIBRERÍA: mknod();

PROTOTIPO: int mknod( char *nombre, mode_t modo, dev_t disp);
  RETURNS: 0 si éxito,
           -1 si error: errno = EFAULT (nombre no válido)
                                EACCES (permiso denegado)
                                ENAMETOOLONG (nombre demasiado largo)
                                ENOENT (nombre no válido)
                                ENOTDIR (nombre no válido)
                                (vea la página mknod(3) para más información)

NOTES: Crea un nodo del sistema de ficheros (fichero, dispositivo, o FIFO)
```

Dejaré una discusión más detallada de mknod() a la página del manual, pero lo podemos considerear un simple ejemplo de la creación de un FIFO en C:

```
mknod("/tmp/MIFIFO", S_IFIFO|0666, 0);
```

En este caso el fichero "/tmp/MIFIFO" se crea como fichero FIFO. Los permisos requeridos son "0666", aunque se ven afectados por la configuración de umask de la siguiente forma:

```
umask_definitiva = permisos_solicitados & ~umask_inicial
```

Un truco común es usar la llamada del sisterma umask() para borrar temporalmente el valor de umask:

```
umask(0);
mknod("/tmp/MIFIFO", S_IFIFO|0666, 0);
```

Además, el tercer argumento de mknod() se ignora salvo que estemos creando un archivo de dispositivo. En ese caso, se debería especificar los números *mayor* y *menor* del fichero de dispositivo.

6.3.3 Operaciones con FIFOs

Las operaciones E/S sobre un FIFOson esencialmente las mismas que para las tuberías normales, con una gran excepción. Se debería usar una llamada del sistema ºpen" o una función de librería para abrir físicamente un canal para la tubería. Con las tuberías semi-duplex, esto es innecesario, ya que la tubería reside en el núcleo y no en un sistemade archivos físico. En nuestro ejemplo trataremos la tubería como un stream, abiéndolo con fopen(), y cerrándolo con fclose().

Consideramos un proceso servidor simple:

```
/*****************************************************************************
 Parte de la "Guia Linux de Programacion - Capitulo 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULO: fifoserver.c
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#include <linux/stat.h>

#define FIFO_FILE       "MIFIFO"

int main(void)
{
        FILE *fp;
        char readbuf[80];

        /* Crea el FIFO si no existe */
        umask(0);
        mknod(FIFO_FILE, S_IFIFO|0666, 0);

        while(1)
        {
                fp = fopen(FIFO_FILE, "r");
                fgets(readbuf, 80, fp);
                printf("Cadena recibida: %s\n", readbuf);
                fclose(fp);
        }

        return(0);
}
```

Como un FIFO bloquea por defecto, ejecute el servidor en segundo plano tras compilarlo:

```
$ fifoserver&
```

Discutiremos la acción de bloqueo de un FIFO en un momento. Primero considrearemos el siguiente cliente simple enfrentado a nuestro servidor:

```
/*****************************************************************************
 Parte de la "Guia Linux de Programacion - Capitulo 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULO: fifoclient.c
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define FIFO_FILE       "MIFIFO"

int main(int argc, char *argv[])
{
        FILE *fp;

        if ( argc != 2 ) {
                printf("USO: fifoclient [cadena]\n");
                exit(1);
        }

        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }

        fputs(argv[1], fp);

        fclose(fp);
        return(0);
}
```

6.3.4 Acciones Bloqueantes en una FIFO

Normalmente, el bloqueo ocurre en un FIFO. En otro palabras, si se abre el FIFO para lectura, el proceso esará "bloqueado" hasta que cualquier otro proceso lo abra para escritura. Esta acción funciona al revés también. Si este comportamiento no nos interesa, se puede usar la bandera O_NONBLOCK en la llamada a open() para desactivar la acción de bloqueo por defecto.

En el caso de nuestro servidor simple, lo hemos puesto en segundo plano, y permito hacer su bloqueo allí. La alternativa estarían saltar a otra consola virtual y ejecutar el cliente, cambiando de un lado a otro para ver la acción resultante.

6.3.5 La Infame Señal SIGPIPE

En una última nota, las tuberías deberían tener a un lector y un escritor. Si un proceso trata de escribir en una tubería que no tiene lector, el núcleo enviará la señal SIGPIPE. Esto es imperativo cuando en la tubería se ven envueltos más dos procesos.

6.4 IPC en Sistema V

6.4.1 Conceptos fundamentales

Con Unix Sistema V, AT&T introdujo tres nuevas formas de las facilidades IPC (colas de mensajes, semáforos y memoria compartida). Mientras que el comité POSIX aun no ha completado su estadarización de estas facilidades, la mayoría de las implementaciones soportan éstas. Además, Berkeley (BSD) usa sockets como su forma primaria de IPC, más que los elementos del Sistema V. Linux tiene la habilidad de usar ambas formas de IPC (BSD y System V), aunque no se discutirán los socket hasta el último capítulo.

La implementación para Linux del IPC System V fue escrita por *Krishna Balasubramanian*, en `balasub@cis.ohio-state.edu`.

Identificadores IPC

Cada *objeto IPC* tiene un único identificador IPC asociado con él. Cuando decimos "objeto IPC", hablamos de una simple cola de mensaje, semáforo o segmento de memoria compartida. Se usa este identificador, dentro del núcleo, para identificar de forma única un objeto IPC. Por ejemplo, para acceder un segmento particular memoria compartida, lo único que requiere es el valor ID único que se le ha asignado a ese segmento.

La unicidad de un identificador es importante según el *tipo* de objeto en cuestión. Para ilustrar esto, supondremos un identificador numérico "12345". Mientras no puede haber nunca dos colas de mensajes, con este mismo identificador existe la posibilidad que existan una cola de mensajes y un segmento de memoria compartida que poseen el mismo identificador numérico.

Claves IPC

Para obtener un identificador único, debe utilizarse una *clave*. Ésta debe ser conocida por ambos procesos cliente y servidor. Este es el primer paso para construir el entorno cliente/servidor de una aplicación.

Cuando usted llama por teléfono a alguien, debe conocer su número. Además, la compañía telefónica debe conocer cómo dirigir su llamada al destino. Una vez que el receptor responde a su llamada, la conexión tiene lugar.

En el caso de los mecanismos IPC de Sistema V, el "teléfono" coincide con el tipo de objeto usado. La "compañía telefónica" o el sistema de encaminado, se puede equiparar con la clave IPC.

La clave puede ser el mismo valor cada vez, incluyendo su código en la propia aplicación. Esta es una desventaja pues la clave requerida puede estar ya en usa. Por eso, la función `ftok()` nos será útil para generar claves no utilizadas para el cliente y el servidor.

```
FUNCIÓN DE LIBRERÍA: ftok();

PROTOTIPO: key_t ftok ( char *nombre, char proj );
  RETORNA: nueva clave IPC si hay éxito
           -1 si no hubo éxito, dejando errno con el valor de la llamada stat()
```

La clave del valor devuelto de ftok () se genera por la combinación del número del inodo y del número menor de dispositivo del archivo argumento, con el carácter identificador del proyecto del segundo argumento. Éste no garantiza la unicidad, pero una aplicación puede comprobar las colisiones y reintentar la generación de la clave.

```
key_t   miclave;
miclave = ftok("/tmp/miaplic", 'a');
```

En el caso anterior el directorio `/tmp/miaplic` se combina con la letra '`a`'.Otro ejemplo común es usar el directorio actual:

```
key_t   miclave;
mykey = ftok(".", 'a');
```

El algoritmo de la generación de la clave usado está completamente a la discreción del programador de la aplicación. Mientras que tome medidas para prevenir ls condiciones críticas, bloqueos, etc, cualquier método es viable. Para nuestros propósitos de demostración, usaremos ftok(). Si suponemos que cada proceso cliente estará ejecutándose desde un único directorio "home", las claves generadas deben bastar por nuestras necesidades.

El valor clave, sin embargo, se obtiene, se usa una llamada al sistema IPC para crear u obtener acceso a los objetos IPC.

Comando `ipcs`

El comando `ipcs` puede utilizarse para obtener el estado de todos los objetos IPC Sistema V. La versión para Linux de esta utilidad también fue preparada por *Krishna Balasubramanian*.

```
ipcs     -q:    Mostrar solo colas de mensajes
ipcs     -s:    Mostrar solo los semáforos
ipcs     -m:    Mostrar solo la memoria compartida
ipcs --help:    Otros argumentos
```

Por defecto, se muestran las tres categorías. Considérese el siguiente ejemplo de salida del comando `ipcs`:

```
------ Shared Memory Segments --------
shmid     owner     perms     bytes     nattch    status

------ Semaphore Arrays --------
semid     owner     perms     nsems     status

------ Message Queues --------
msqid     owner     perms     used-bytes  messages
0         root      660       5           1
```

Aquí vemos una simple cola mensaje que tiene un identificador " 0." Es propiedad del `root`, y tiene permisos en octal de `660`, o `-rw-rw--`. Hay un mensaje en la cola, y ese mensaje tiene un tamaño del total de 5 bytes.

Los comandos `ipcs` son una herramienta muy potente que proporciona una leve introducción en los mecanismos de almacenamiento del núcleo para objetos IPC. Apréndalo, úselo, reverencielo.

El Comando `ipcrm`

Se puede usar el comando ipcrm para quitar un objeto IPC del núcleo. Mientras que los objetos IPC se pueden quitar mediante llamadas al sistema en el código del usuario (veremos cómo en un momento),aparece a menudo la necesidad, sobre todo en ambientes del desarrollo, de quitar objetos IPC a mano. Su uso es simple:

```
ipcrm <msg | sem | shm>  <IPC ID>
```

Simplemente especifique si el objeto a eliminar es una cola de mensaje (em msg), un semáforo (*sem*), o un segmento de memoria compartida (*shm*). El identificador de IPC se puede obtenr mediante los comandos ipcs. Tiene que especificar el tipo de objeto, como los identificadores son únicos entre los del mismo tipo (retome nuestra discusión anterior).

6.4.2 Colas de Mensajes

Conceptos Básicos

Las colas de mensaje se pueden describir mejor como una lista enlazada interior dentro del espacio de direccionamiento del núcleo.Los mensajes se pueden enviar a la cola en orden y recuperarlos de la cola en varias maneras diferentes. Cada cola de mensaje (por supuesto) está identificada de forma única por un identificador IPC.

Estructuras interna y de datos de usuario

La clave para comprender totalmente tales temas complejos como el IPC Sistema V es familiarizarse con las distintas estrcturas de datos internas que residen dentro de los confines del núcleo mismo.El acceso directo a algunas de estas estructuras es necesario incluso en las operacions más primitivas, mientras otros residen a un nivel mucho más bajo.

Buffer de Mensaje La primera estructura que veremos es la estructura `msgbuf`. Esta particular estructura de datos puede ser interpretada como una *plantilla* por datos del mensaje. Mientras que un programador puede elegir si definir estructuras de este tipo, es imperativo que entiende que **hay** realmente una estructura del tipo `msgbuf`. Se declara en `linux/msg.h` como sigue:

```
/* buffer de mensaje para llamadas msgsnd y msgrcv */
struct msgbuf {
    long mtype;         /* tipo de mensaje */
    char mtext[1];      /* texto del mensaje */
};
```

Hay dos miembros en la estructura `msgbuf`:

`mtype`

El *tipo* de mensaje, representado por un número positivo. ¡Y *debe* ser un número positivo!

`mtext`

Los datos del mensaje en sí mismo.

La habilidad asignarle a un mensaje dado un *tipo*, esencialmente le da la capacidad de *multiplexar* mensajes en una cola sola. Por ejemplo, al proceso cliente se puede asignar a un número mágico, que se puede usar como el tipo de mensaje para mensajes enviados desde un proceso servidor. El servidor mismo podría usar algunos otros números, que los clientes podrían usar para enviarle mensajes. Por otra parte, una aplicación podría marcar mensajes de error como tener un mensaje tipo 1, petición de mensajes podrían ser tipo 2, etc. Las posibilidades son interminables.

En otra nota no se confunda por el nombre demasiado descriptivo asignado al elemento dato del mensaje (`mtext`). Este campo no se restringe a contener sólo arrays de carácteres, sino cualquier tipo e dato, en cualquier forma. El campo mismo es realmente arbitrario, ya que esta estructura es redefinida por el programador de la aplicación. Considere esta redefinición:

```
struct my_msgbuf {
        long    mtype;          /* Tipo de mensaje */
        long    request_id;     /* Identificador de petición */
        struct  client info;    /* Estructura de información del cliente */
};
```

Aquí vemos el tipo de mensaje, como antes, pero el resto de la estructura ha sido reemplazado por otros dos elementos, uno de los cuales es otra estructura. Ésta es la belleza de las colas de mensajes. El núcleo no hace ninguna traducción de datos. Se puede enviar cualquier información.

Sin embargo, existe un límite interior del máximo tamaño de un mensaje dado. En Linux se define éste en `linux/msg.h` como sigue:

```
#define MSGMAX  4056   /* <= 4056 */   /* Tama~no maximo del mensaje (bytes) */
```

El mensajes no puede ser mayor de 4,056 bytes en total, incluyendo el miembro mtype, que tiene una longitud de 4 bytes (long).

Estructura `msg` del Núcleo El núcleo guarda cada mensaje en la cola dentro de la estructura msg. Se define en `linux/msg.h` como sigue:

```
/* one msg structure for each message */
struct msg {
    struct msg *msg_next;   /* siguiente mensaje de la cola */
    long  msg_type;
    char *msg_spot;         /* direccion del texto del mensaje */
    short msg_ts;           /* tama~no del texto del mensaje */
};
```

`msg_next`

Es un puntero al siguiente mensaje de la cola. Se almacenan como ouna lista simple enlazada en el espacio de direcciones del núcleo.

`msg_type`

Éste es el tipo de mensaje, como asignado en la estructura msgbuf del usuario.

`msg_spot`

Un puntero al inicio del cuerpo del mensaje.

`msg_ts`

La longitud del texto del mensaje o del cuerpo.

Estructura `msqid_ds` del núcleo Cada uno de los tres tipos de objetos IPC tienen una estructura de datos interna que se mantiene por el núcleo. Para las colas de mensaje, es la estructura msqid_ds. El núcleo crea, almacena, y mantiene un caso de esta estructura por cada cola de mensaje hace cola creada en el sistema. Se define en `linux/msg.h` de la siguiente forma:

```
/* una estructura msqid por cada cola del sistema */
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first;  /* primer mensaje de la cola */
    struct msg *msg_last;   /* ultimo mensaje */
    time_t msg_stime;       /* ultimo instante de msgsnd */
    time_t msg_rtime;       /* ultimo instante de msgrcv */
    time_t msg_ctime;       /* ultimo instante cambio */
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    ushort msg_cbytes;
    ushort msg_qnum;
    ushort msg_qbytes;      /* numero maximo de bytes en cola */
    ushort msg_lspid;       /* pid del ultimo msgsnd */
    ushort msg_lrpid;       /* pid de la ultima recepcion */
};
```

Aunque tendrá raramente que usar la mayor parte de los miembros de esta estructura, daremos una descripción breve de está para completar nuestra visión:

`msg_perm`

Un caso de la estructura `ipc_perm`, que se define en `linux/ipc.h`. Éste recoge la información del permiso para la cola de mensaje, incluso los permisos del acceso, e información sobre el creador de la cola (uid, etc).

`msg_first`

Enlace al primer mensaje de la cola (cabecera de la lista).

`msg_last`

Enlace al último mensaje de la cola (cola de la lista).

`msg_stime`

Instante (`time_t`) del último mensaje que se envió a la cola.

`msg_rtime`

Instante del último mensaje recuperado de la cola.

`msg_ctime`

Instantedel último cambio hecho a la cola. (hablaremos de esto más tarde).

`wwait`

y

`rwait`

Punteros a la *cola de espera* del núcleo. Se usan cuando una operación sobre una cola de mensajes estima que el proceso entra en el estado de dormido (es decir, la cola está llena y el porceso espera una apertura).

`msg_cbytes`

Número total number de bytes que hay en la cola (suma de los tamaños de todos los mensajes).

`msg_qnum`

Número de mansajes actual en la cola.

`msg_qbytes`

Máximo número de bytes en la cola.

`msg_lspid`

El PID del proceso que envía el último mensaje.

`msg_lrpid`

El PID del proceso que recupera el último mensaje.

Estructura `ipc_perm` del núcleo El núcleo guarda información de permisos para objetos IPC en una estructura de tipo `ipc_perm`. Por ejemplo, en la estructura interna para una cola de mensaje descrita antes, el miembro de `msg_perm` es de este tipo. Se declara en `linux/ipc.h` como sigue:

```
struct ipc_perm
{
  key_t  key;
  ushort uid;   /* euid y egid del propietario */
  ushort gid;
  ushort cuid;  /* euid y egid del creador */
  ushort cgid;
  ushort mode;  /* modos de acceso, veanse despues los valores */
  ushort seq;   /* numero de secuencia del slot */
};
```

Todo lo anterior es bastante autoexplicativo. Guardado junto con la clave IPC del objeto hay información sobre el creador y dueño del objeto (pueden ser diferentes).Los modos del acceso octal como un `unsigned short`. Finalmente, la *secuencia del slot* se guarda al final. Cada vez qye un objeto IPC se cierra mediante una llamada al sistema llama (destruye), este valor se incrementa por el máximo númera de objetos IPC que pueden residir en un sistema. ¿Tendrá que usar este valor? No.

NOTA:*Hay una excelente exposición de este tema, y los asuntos de seguridad relacionados, en el libro* **UNIX Network Programming**, *de Richard Stevens (página 125).*

LLAMADA AL SISTEMA: msgget()

Para crear una nueva cola de mensajes, o acceder a una existente, usaremos la llamada al sistema `msgget()`.

```
LLAMADA AL SISTEMA: msgget();

PROTOTIPO: int msgget ( key_t clave, int msgflg );
  RETORNA: Si hay éxito, identificador de la cola de mensajes
           -1 si error: errno = EACCESS (permiso denegado)
                                EEXIST (No puede crearse la cola pues ya existe)
                                EIDRM (La cola esta marcada para borrarse)
                                ENOENT (La cola no existe)
                                ENOMEM (No hay memoria para crear la cola)
                                ENOSPC (Se ha superado el limite de colas)
NOTAS:
```

El primer argumento de `msgget()` es el valor clave (en nuestro caso devuelto por una llamada a `ftok()`. Este valor clave se compara entonces con los valores clave que existen dentro del núcleo de otras colas de mensaje. En ese punto las operaciones de apertura o acceso depende de los contenidos del argumento `msgflg`.

IPC_CREAT

Crea la cola si aun no existe en el núcleo.

IPC_EXCL

Cuando se usa con IPC_CREAT, falla si la cola ya existe.

Si usamos solo `IPC_CREAT`, `msgget()` retornará el identificador de una cola nueva, o bien el de la existente con la misma clave. Si usamos además `IPC_EXCL`, la llamada creará una nueva cola o fallará si la cola con esa clave ya existía. La opción `IPC_EXCL` es poco útil si no se usa combinada con `IPC_CREAT`.

Es posible incluir en la máscara un modo opcional octal, pues cada objeto IPC tiene un esquema de permisos de acceso similar a cualquier archivo del sistema Unix.

Creamos una función de envoltura rápida para abriro crear una cola de mensaje:

```
int abrir_cola( key_t keyval )
{
        int     qid;

        if((qid = msgget( keyval, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }
```

```
        return(qid);
}
```

Nótese el uso del modo de permisos `0660`. Esta pequeña función retornará, bien un identificador entero de la cola de mensajes, o -1 si hubo error. El valor de la clave (keyval) debe ser el único argumento de la llamada a la función.

LLAMADA AL SISTEMA: msgsnd()

Una vez que tenemos el identificador de la cola, podemos empezar a realizar operaciones sobre ella. Para entregar un mensaje a una cola, use la llamada al sistema `msgsndl`:

```
  LLAMADA AL SISTEMA: msgsnd();

  PROTOTIPO: int msgsnd ( int msqid, struct msgbuf *msgp, int msgsz, int msgflg );
    RETORNA: 0 si éxito
             -1 si error: errno = EAGAIN (la cola está llena, y se usó IPC_NOWAIT).
                                  EACCES (permiso denegado, no se puede escribir)
                                  EFAULT (Dirección de memoria de msgp inválida)
                                  EIDRM  (La cola de mensajes fue borrada)
                                  EINTR  (Se recibió una señal mientras se esperaba para escribir)
                                  EINVAL (Identificador de cola inválido, tipo
                                          no positivo o tamaño de mensaje inválido)
                                  ENOMEM (No hay memoria suficiente para copiar el buffer)
  NOTAS:
```

El primer argumento de `msgsnd` es nuestro identificador de la cola, devuelto por un llamada previa a `msgget`. El segundo argumento, `msgp`, es un puntero a nuestro buffer redeclarado y cargado. El argumento msgsz contiene el tamaño del mensaje en bytes, excluye la longitud del tipo de mensaje (4 byte).

El argumento `msgflg` se puede poner a cero (ignorado), o:

IPC_NOWAIT

Si la cola del mensaje está llena, entonces no se escribe en la cola el mensaje, y se le devuelve el control la proceso llamador. Si no se especifica, entonces el proceso llamador se suspenderá (bloqueado) hasta que se puede escribir el mensaje.

Creamos otra función de la envoltura por enviar mensajes:

```
int enviar_msj( int qid, struct mymsgbuf *qbuf )
{
        int     resultado, longitud;

        /* La longitud es esencialmente el tama˜no de la estructura menos sizeof(mtype) */
```

```
        longitud = sizeof(struct mymsgbuf) - sizeof(long);

        if((resultado = msgsnd( qid, qbuf, length, 0)) == -1)
        {
                return(-1);
        }

        return(resultado);
}
```

Esta pequeña función intenta enviar un mensaje almacenado en la dirección pasada (`qbuf`) a la cola de mensajes identificada por el número pasado en el argumento `qid`. Aquí tenemos un programa de ejemplo que utiliza las dos funciones que hemos desarrollado aquí:

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/ipc.h>
#include <linux/msg.h>

main()
{
        int    qid;
        key_t  msgkey;
        struct mymsgbuf {
                long    mtype;          /* Tipo de mensaje */
                int     request;        /* Numero de trabajo */
                double  salary;         /* Salario del empleado */
        } msg;

        /* Generamos nuestra clave IPC */
        msgkey = ftok(".", 'm');

        /* Abrir/crear la cola */
        if(( qid = abrir_cola( msgkey)) == -1) {
                perror("abrir_cola");
                exit(1);
        }

        /* Preparar mensajes con datos arbitrarios */
        msg.mtype   = 1;        /* !El mensaje debe ser numero positivo! */
        msg.request = 1;        /* Dato numero 1 */
        msg.salary  = 1000.00;  /* Dato numero 2 (!mi salario anual!) */

        /* !Bombear mensaje! */
        if((enviar_msj( qid, &msg )) == -1) {
                perror("enviar_msj");
```

```
                exit(1);
        }
}
```

Tras crear/abrir la cola de mensajes, pasamos a preparar el buffer del mensaje con datos de prueba (*note la falta de datos de tipo carácter para ilustrar nuestro punto sobre envío de información binaria*). Una simple llamada a `enviar_msj` envía nuestro mensaje a la cola.

Ahora que tenemos un mensaje en la cola, probemos en comando `ipcs` para comprobar el estado de ésta. Ahora continuaremos con la discusión para ver cómo leer información del mensaje. Para ello, se utiliza la llamada al sistema `msgrcv()`:

```
  LLAMADA AL SISTEMA: msgrcv();
  PROTOTIPO: int msgrcv ( int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg );
    RETURNS: Número de bytes copiados en la cola de mensajes
             -1 si error: errno = E2BIG  (La longitud del mensaje es mayor que msgsz)
                                  EACCES (No hay permiso de lectura)
                                  EFAULT (La dirección del buffer msgp es incorrecta)
                                  EIDRM  (La cola fue eliminada durante la lectura)
                                  EINTR  (Interrumpido por llegada de señal)
                                  EINVAL (msgqid inválida, o msgsz menor que 0)
                                  ENOMSG (IPC_NOWAIT incluido, y no hay mensajeen la
                                          cola disponible para leer)
  NOTAS:
```

Obviamente, el primer argumento se usa para especificar la cola utilizada durante el proceso de recuperación del mensaje (se debería haber sido devuelto por una llamada anterior a `msgget`). El segundo argumento (`msgp`) representa la dirección de una variable buffer de mensaje para guardar el mensaje recuperado. El tercer argumento, (`msgsz`), representa el tamaño de la estructura del buffer del mensaje, excluye la longitud del miembro de `mtype`. Una vez más, se puede calcular éste fácilmente como:

```
msgsz = sizeof(struct mymsgbuf) - sizeof(long);
```

El cuarto argumento (`mtype`) especifica el *tipo* de mensaje a recuperar de la cola. El núcleo buscará la cola por el último mensaje que cuadra con el tipo, y le devolverá a una copia de él en la dirección apuntada a por el argumento `msgp`. Existe un caso especial. Si se pasa el argumento `mtype` con un valor de ceros, entonces se devuelve el mensaje más viejo en la cola, independiente del tipo.

Si se pasa como una bandera **IPC_NOWAIT**, y no hay ningún mensajes disponibles, la llamada le devuelve `ENOMSG` al proceso llamador. Por otra parte, el proceso llamador se bloquea hasta que un mensaje llega a la cola que satisface el parámetro **`msgrcv()`**. Si se anula la cola mientras un cliente espera en un mensaje, se devuelve `EIDRM`. Se devuelve `EINTR` si se coge una señal mientras el proceso está en medio del bloqueo, y espera la llegada de un mensaje.

Examinamos una función de envoltura rápida para recuperar un mensaje de nuestra cola:

```
int leer_msj( int qid, long type, struct mymsgbuf *qbuf )
{
        int     resultado, longitud;

        /* La longitud es esencialmente el tama~no del buffer menos sizeof(long) */
        longitud = sizeof(struct mymsgbuf) - sizeof(long);

        if((resultado = msgrcv( qid, qbuf, length, type,  0)) == -1)
        {
                return(-1);
        }

        return(resultado);
}
```

Después de terminar de forma efectiva la recuperación de un mensaje en la cola, se destruye la entrada del mensaje dentro de la cola.

El bit **MSG_NOERROR** del argumento `msgflg` proporciona algunas capacidades adicionales. Si el tamaño de los datos del mensaje físico es mayor que `msgsz`, y **MSG_NOERROR** está indicado, entonces se trunca el mensaje, y se devuelven sólo msgsz bytes. Normalmente, la llamada al sistema `msgrcv()` devuelve -1 (**E2BIG**), y el mensaje quedará en la cola para una recuperación posterior. Esta conducta se puede usar para crear otra función de envoltura, que nos permitirá "mirar" en la cola, para ver si un mensaje ha llegado y satisface nuestra demanda, sin sacarlo realmente de la cola:

```
int mirar_msj( int qid, long type )
{
        int     resultado, longitud;

        if((resultado = msgrcv( qid, NULL, 0, type,  IPC_NOWAIT)) == -1)
        {
                if(errno == E2BIG)
                        return(TRUE);
        }

        return(FALSE);
}
```

Arriba, se dará cuenta de la falta de una dirección de buffer y una longitud. En este caso particular *queremos* que la llamada falle. Sin embargo, verificamos por el retorno de **E2BIG** que indica que existe un mensa del tipo de nuestra petición. La función de envoltura vuelve **TRUE** en éxito, **FALSO** en otro caso. También observa el uso de **IPC_NOWAIT**, que previene el compoeramiento de bloque visto antes.

LLAMADA AL SISTEMA: msgctl()

Por el desarrollo de las funciones de envoltura anteriores, ahora tiene una aproximación simple y elegante para crear y utilizar las estructuras internas asociadas con colas de mensaje en sus aplicaciones. Ahora, volveremos directamente a la discusión sobre la manipulación de las estructuras internas asociadas con una colas de mensaje dada.

Para realizar operaciones de control en una cola de mensaje, use la llamada al sistema `msgctl()`.

```
LLAMADA AL SISTEMA: msgctl();
PROTOTIPO: int msgctl ( int msgqid, int cmd, struct msqid_ds *buf );
  RETORNA: 0 si éxito
           -1 si error: errno = EACCES (No hay permiso de lectura y cmd vale IPC_STAT)
                                EFAULT (Direccion de buf inválida con los comandos IPC_SET y
                                        IPC_STAT)
                                EIDRM  (La cola fue eliminada durante la operación)
                                EINVAL (msgqid inválida, o msgsz menor que 0)
                                EPERM  (Se intentó el comando IPC_SET o IPC_RMID, pero
                                        no tiene acceso de escritura (alteración)
                                        de la cola)
NOTAS:
```

Ahora, el sentido común dice que la manipulación directa de las estructuras de datos internas del núcleo podría ocasionar alguna juerga nocturna. Desgraciadamente, los deberes resultantes por parte del programador se podrían clasificar como diversión sólo si gusta desecha el subsistema IPC. Usando msgctl() con un conjunto selectivo de órdenes, tiene la posibilidad de manipular esos elementos, que es menos probable que causen problemas. Echemos un vistazo a estos comandos:

IPC_STAT

Recupera la estructura msqid_ds para una cola, y, la en la dirección del argumento buff.

IPC_SET

Pone el valor del miembro ipc_perm de la estructura msqid_ds para la cola. Toma los valores del argumento buf.

IPC_RMID

Borra la cola del núcleo.

Retomamos nuestra discusión sobre las estructuras de datos internas para colas de mensaje (`msqid_ds`). El núcleo mantiene una instancia de esta estructura por cada cola que existe en el sistema. Usando el comando **IPC_STAT**, podemos recuperar una copia de esta estructura para examinarla. Miramos una función de envoltura rápida que recuperará la estructura interna y la copia en una dirección pasada:

```
int leer_queue_ds( int qid, struct msgqid_ds *qbuf )
{
        if( msgctl( qid, IPC_STAT, qbuf) == -1)
        {
                return(-1);
        }

        return(0);
}
```

Si no podemos copiar el buffer interno, se devuelve -1 a la función que hizo la llamadda. Si todo fue bien, se devuelve un valor 0 (cero), y el buffer pasado debe contener una copia de la estructura de datos interna para la cola representada por el identificador de cola pasado (`qid`).

¿Ahora que tenemos una copia de las estructura de datos interna de una cola, qué se puede manipular, y cómo se puede alterar? El único elemento modificable en la estructura de los datos es el miembro `ipc_perm`. Éste contiene los permisos para la cola, así como información sobre el dueño y creador. Sin embargo, los únicos miembros de la estructura `ipc_perm` que son modificables son modo, uid, y gid. Puede cambiar el id del usuario del dueño, el id del grupo del dueño, y los permisos del acceso para la cola.

Creamos una función de envoltura diseñada para cambiar el modo de una cola. Se debe pasar el modo en como un array de caracteres (por ejemplo *"660"*).

```
int cambiar_modo_cola( int qid, char *modo )
{
        struct msqid_ds tmpbuf;

        /* Obtener copia de la actual estructura de datos interna */
        leer_queue_ds( qid, &tmpbuf);

        /* Cambiar los permisos usando un viejo truco */
        sscanf(mode, "%ho", &tmpbuf.msg_perm.mode);

        /* Actualizar la estructura de datos interna */
        if( msgctl( qid, IPC_SET, &tmpbuf) == -1)
        {
                return(-1);
        }
```

```
        return(0);
}
```

Recuperamos una copia de la estructura de datos interna actual mediante una rápida llamada a nuestra función de envoltura `leer_queue_ds`. Entonces hacemos una llamada a `sscanf()` para alterar el miembro modo de la estructura msg_perm asociada. Sin embargo, no se producen cambios hasta que se usa la nueva copia para actualizar la versión interna. Esto es ejecutado mediante una llamada a `msgctl()` usando el comando el IPC_SET.

¡TENGA CUIDADO! ¡Es posible alterar los permisos en una cola, y al hacerlo, puede cerrarse sin darse cuenta. Recuerde, estos objetos IPC no se van a menos que se quiten propiamente, o el sistema se reinicie. Así, aun cuando no pueda ver una cola con ipcs no significa que no esté allí.

> *Para ilustrar este punto, una anécdota algo cómica parece estar a punto. Mientras daba una clase de enseñanza sobre UNIX interno en la Universidad de Florida Sur, tropecé con un bloque bastante penoso. Había marcado en el servidor del laboratorio la noche de antes, para compilar y probar el trabajo de la clase de la semana. En el proceso de comprobación, me dí cuenta de que había hecho un typo en la codificación para alterar los permisos en una cola de mensaje. Creé una cola simple de mensaje, y probé el envío y la recepción problemas. ¡Sin embargo, cuando intenté cambiar el modo de la cola de "660" a "600", la acción resultante fue que se cerró con llave fuera de mi propia cola! Como un resultado, no podía probar la cola de mensaje en la misma área de mi directorio de fuente. Entonces usé la función ftok() para crear el código IPC codifica, trataba de acceder a una cola para la que no tenía permisos. Acabé y me puse en contacto con el administrador del sistema local en la mañana siguiente, perdiendo una hora para explicarle a él lo que era una cola del mensaje, y porqué necesitaba correr los comandos ipcrm.*

Después de recuperar con éxito un mensaje de la cola, se quita el mensaje. Sin embargo, como mencioné antes, los objetos IPC quedan en el sistema a menos que se quiten explícitamente, o el sistema sea reiniciado. Por consiguiente, nuestra cola de mensaje hace cola todavía existe dentro del núcleo, disponible para usarla después de que sólo un simple mensaje desaparezca. Para completar el ciclo de vida de una cola de mensaje, se deben quitar con una llamada a `msgctl()`, usando el comando **IPC_RMID**:

```
int borrar_cola( int qid )
{
        if( msgctl( qid, IPC_RMID, 0) == -1)
        {
                return(-1);
```

```
        }

        return(0);
}
```

Esta función devuelve 0 si la cola se borró sin problemas, o si no devuelve un valor -1. El borrado de la cola es atómico, y cualquier acceso posterior a ella para cualquier cosa fallará.

msgtool: Un manipulador interactivo de colas de mensajes.

Pocos pueden negar el beneficio inmediato de tener información técnica exacta rápidamente disponible. Tales materiales proporcionan un mecanismo tremendo para el aprendizaje y la exploración de áreas nuevas. En la misma nota, teniendo ejemplos reales para acompañar cualquier información técnica, acelerará y reforzará el proceso de aprendizaje.

Hasta ahora, los únicos ejemplos útiles que se han presentado eran las funciones de envoltura para manipular colas de mensaje. Aunque son sumamente útiles, no se han presentado en una manera que garantice su estudio y experimentación. Esto se solucionará con *msgtool*, una utilidad de la línea de comando interactiva parar manipular colas de mensaje IPC. Mientras funciona como una herramienta adecuada para refuerzo de la educación, también se puede aplicar directamente en asignaciones reales, para proporcionar la funcionalidad de las colas de mensaje mediante script de shell normales.

Vistazo rápido El programa `msgtool` cuenta con argumentos de la línea del comando para determinar su comportamiento. Éste es lo que lo hace especialmente útil cuando es llamado desde script de shell. Se proporcionan todas las capacidades, de crear, enviar, y recuperar, a cambiar los permisos y finalmente eliminar una cola. Actualmente, usa un array de caracteres para los datos, permitiéndole enviar mensajes textuales. Los cambios para facilitar tipos de datos adicionales se queda como un ejercicio para el lector.

Sintaxis de la línea de comandos

Envío de mensajes

```
msgtool e (tipo) "texto"
```

Recepción de Mensajes

```
msgtool r (tipo)
```

Cambio de los permisos

```
msgtool m (modo)
```

Borrado de una cola

```
msgtool d
```

Ejemplos

```
msgtool  e   1 prueba
msgtool  e   5 prueba
msgtool  e   1 "Esto es una prueba"
msgtool  r   1
msgtool  b
msgtool  m   660
```

Código Fuente Seguidamente ofrecemos el código fuente de la utilidad `msgtool`. Debe compilar sin problemas con cualquier revisión (decente) del núcleo que soporte IPC Sistema V. ¡Asegúrese de activar el IPC en el núcleo cuando lo recompile!

Como comentario, esta utilidad *creará* una cola de mensajes si no existe, independientemente del tipo de acción solicitado.

> **NOTA:** *Puesto que esta utilidad usa ftok() para generar claves IPC, pueden encontrarse conflictos de directorios. Si cambia de directorio durante la ejecución de su script, posiblemente no funcione bien. Otra solución sería codificar dentro del programa* `msgtool` *un path completo a la utilidad, tal como "/tmp/msgtool", o bien pasarle dicho path mediante un nuevo argumento de la línea de comandos.*

```
/*****************************************************************************
 Parte de la "Guia Linux de Programacion - Capitulo 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULO: msgtool.c
 *****************************************************************************
 Utilidad de manejo de las colas de mensajes del sistema IPC SYSV
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_SEND_SIZE 80

struct mymsgbuf {
```

```
        long mtype;
        char mtext[MAX_SEND_SIZE];
};

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
void read_message(int qid, struct mymsgbuf *qbuf, long type);
void remove_queue(int qid);
void change_queue_mode(int qid, char *mode);
void usage(void);


int main(int argc, char *argv[])
{
        key_t key;
        int   msgqueue_id;
        struct mymsgbuf qbuf;

        if(argc == 1)
                usage();

        /* Crear clave unica mediante ftok() */
        key = ftok(".", 'm');

        /* Abrir la cola -- crearla si es necesario */
        if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1) {
                perror("msgget");
                exit(1);
        }

        switch(tolower(argv[1][0]))
        {
                case 'e': send_message(msgqueue_id, (struct mymsgbuf *)&qbuf,
                                          atol(argv[2]), argv[3]);
                          break;
                case 'r': read_message(msgqueue_id, &qbuf, atol(argv[2]));
                          break;
                case 'b': remove_queue(msgqueue_id);
                          break;
                case 'm': change_queue_mode(msgqueue_id, argv[2]);
                          break;

                 default: usage();

        }

        return(0);
}
```

```
void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
{
        /* Enviar mensaje a la cola */
        printf("Enviando mensaje ...\n");
        qbuf->mtype = type;
        strcpy(qbuf->mtext, text);

        if((msgsnd(qid, (struct msgbuf *)qbuf,
                strlen(qbuf->mtext)+1, 0)) ==-1)
        {
                perror("msgsnd");
                exit(1);
        }
}

void read_message(int qid, struct mymsgbuf *qbuf, long type)
{
        /* Leer mensaje de la cola */
        printf("Leyendo mensaje ...\n");
        qbuf->mtype = type;
        msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);

        printf("Tipo: %ld Texto: %s\n", qbuf->mtype, qbuf->mtext);
}

void remove_queue(int qid)
{
        /* Borrado de la cola */
        msgctl(qid, IPC_RMID, 0);
}

void change_queue_mode(int qid, char *mode)
{
        struct msqid_ds myqueue_ds;

        /* Obtener informacion actual */
        msgctl(qid, IPC_STAT, &myqueue_ds);

        /* Convertir y cargar el modo */
        sscanf(mode, "%ho", &myqueue_ds.msg_perm.mode);

        /* Actualizar el modo */
        msgctl(qid, IPC_SET, &myqueue_ds);
}

void usage(void)
```

```
{
        fprintf(stderr, "msgtool - Utilidad de manejo de colas de mensajes\n");
        fprintf(stderr, "\nUSO: msgtool (e)nviar <tipo> <texto>\n");
        fprintf(stderr, "               (r)ecibir <tipo>\n");
        fprintf(stderr, "               (b)orrar\n");
        fprintf(stderr, "               (m)odo <modo octal>\n");
        exit(1);
}
```

6.4.3 Semáforos

Conceptos Básicos

Los semáforos se pueden describir mejor como contadores que se usan para controlar el acceso a recursos compartidos por múltiples procesos. Se usan con más frecuencia como un mecanismo de cierre para prevenir que los procesos accedan a un recurso particular mientras otro proceso lo está utilizando. Los semáforos son a menudo considerados como el más difícil asir de los tres tipos de objetos Sistema V IPC. Para comprender totalmente los semáforos, los discutiremos brevemente antes de comenzar cualquier llamada al sistema y teoría operacional.

El nombre de *semáforo* es realmente un término viejo del ferrocarril, que se usaba para prevenir, en las travesías el cruce en las vías de los viejos carros. Exactamente lo mismo se puede decir sobre un semáforo. Si el semáforo está *abierto* (los brazos en alto), entonces un recurso está disponible (los carros cruzarían las vías). Sin embargo, si el semáforo está *cerrado* (los brazos están abajo), entonces recursos no están disponible (los carros deben esperar).

Mientras que con este este ejemplo simple nos introduce el concepto, es importante darse cuenta de que los semáforos se llevan a cabo realmente como *conjuntos*, en lugar de como entidades solas. Por supuesto, un conjunto de semáforos dado puede tener sólo un semáforo, como en nuestro ejemplo del ferrocarril.

Quizás otra aproximación al concepto de semáforos, sería pensar en ellos como *contadores de recursos*. Apliquemos este concepto a otro caso del mundo real. Considere un *spooler* de impresión, capaz de manipular impresoras múltiples, con cada manejo de la impresora con demandas de la impresión múltiples. Un hipotético manejador del *spool* de impresión utilizará un conjunto de semáforos para supervisar el acceso a cada impresora.

Suponemos que en nuestro cuarto de la impresora de la organización, tenemos 5 impresoreas conectadas. Nuestro manejador del spool asigna un conjunto de 5 semáforos a él, uno por cada impresora del sistema. Como cada impresora es capaz de imprimir físicamente un único trabajo en un instante, cada uno de nuestros cinco semáforos de nuestro conjunto se inicializará a un valor de 1 (uno), lo que significa que están todas en línea, y aceptan trabajos.

Juan envía que una petición de impresión al spooler. El manejador de la impresión mira los semáforos, y encuentra que el primer semáforo que tiene un valor de uno. Ante enviar la petición de Juan al aparato físico, el manejador de impresión *decrementa* el semáforo de la impresora correspondiente con un valor negativo (-1). Ahora, el valor de ese semáforo es cero. En el mundo de semáforos Sistema V, un valor de cero representa el 100ese semáforo. En nuestro ejemplo se no se puede enviar a esa impresora ninguna otra petición hasta que sea distinto de cero.

Cuando el trabajo de Juan ha finalizado, el gestor de impresión *incrementa* el varlor del semáforo correspondiente. Su valor vuelve a ser uno (1), lo que indica que el recurso vuelve a estar disponible. Naturalmente, si los cinco semáforos tienen valor cero, indica que todas las impresoras están ocupadas con peticiones y no se pueden atender más.

Aunque este es un ejemplo simple, procure no confundirse con el valor inicial (1) dado a los semáforos. En realidad, cuando se ven como contadores de recursos, pueden ser iniciados con *cualquier valor positivo*, y no están limitados a valer 0 ó 1. Si las impresoras de nuestro ejemplo fuesen capaces de aceptar 10 trabajos de impresión, habríamos iniciado sus semáforos con el valor 10, decrementándolo en 1 cada vez que llega un trabajo nuevo y reincrementándolo al terminar otro. Como descubriremos en este capítulo, el funcionamiento de los semáforos tiene mucha relación con el sistema de memoria compartida, actuando como *guardianes* para evitar múltiples escrituras en la misma zona de memoria.

Antes de entrar en las llamadas al sistema relacionadas, repasemos varias estructuras de datos internas usadas en las operaciones con semáforos.

Estructuras de datos internas

Veamos brevemente las estructuras de datos mantenidas por el núcleo para los conjuntos de semáforos.

Estructura `semid_ds` del núcleo Como en las colas de mensajes, el núcleo mantiene unas estructuras de datos internas especiales por cada conjunto de semáforos dentro de su espacio de direcciones. Esta estructura es de tipo `semid_ds` y se define en `linux/sem.h` como sigue:

```
/* Hay una estructura semid_ds por cada juego de semáforos */
struct semid_ds {
        struct ipc_perm sem_perm;       /* permisos .. ver ipc.h */
        time_t          sem_otime;      /* ultimo instante semop */
        time_t          sem_ctime;      /* ultimo instante de cambio */
        struct sem      *sem_base;      /* puntero al primer
                                           semaforo del array */
        struct wait_queue *eventn;
        struct wait_queue *eventz;
        struct sem_undo  *undo;         /* deshacer peticiones
                                           del array*/
        ushort          sem_nsems;      /* no. de semaforos del array */
};
```

Como con las colas de mensaje, las operaciones con esta estructura son ejecutados por llamadas especiales al sistema especial, y no se deben usar directamente. Aquí tenemos las descripciones de los campos más interesantes:

`sem_perm`

Este es un caso de la estructura `ipc_perm`, que se define en `linux/ipc.h`. Toma la información de los permisos para el conjunto de semáforos, incluyendo permisos de acceso e información sobre el creador del conjunto (uid, etc).

`sem_otime`

Instante de la última operación `semop()` (un poco más de esto dentro de un momento)

`sem_ctime`

Instante del último cambio de modo

`sem_base`

Puntero al primer semáforo del array (ver siguiente estructura)

`sem_undo`

Número de solicitudes de deshacer en el array (un poco más dentro de un momento)

`sem_nsems`

Número de semáforos en el conjunto (el array)

Estructura `sem` del núcleo En la estructura `semid_ds`, hay un puntero a la base del array del semáforo. Cada miembro del array es del tipo estructura sem. También se define en `linux/sem.h`:

```
/* Una estructura por cada juego de semaforos */
struct sem {
        short   sempid;         /* pid de ultima operacion */
        ushort  semval;         /* valor actual */
        ushort  semncnt;        /* num. procesos esperando
                                   para incrementarlo */
        ushort  semzcnt;        /* num. procesos esperando
                                   que semval sea 0 */
};
```

`sem_pid`

El PID (identificador del proceso) que realizó la última operación `sem_semval`

Valor actual del semáforo

`sem_semncnt`

Número de procesos esperando la disponibilidad del recurso

`sem_semzcnt`

Número de procesos esperando la disponibilidad 100

LLAMADA AL SISTEMA: semget()

Se usa para crear un nuevo conjunto o acceder a uno existente.

```
LLAMADA AL SISTEMA: semget();

PROTOTIPO: int semget ( key_t key, int nsems, int semflg );
  RETORNA: Identificador IPC del conjunto, si éxito
           -1 si error: errno = EACCESS (permiso denegado)
                                EEXIST (no puede crearse pues existe (IPC_EXCL))
                                EIDRM (conjunto marcado para borrarse)
                                ENOENT (no existe el conjunto ni se
                                        indicó IPC_CREAT)
                                ENOMEM (No hay memoria suficiente para crear)
                                ENOSPC (Limite de conjuntos excedido)
NOTAS:
```

El primer argumento de `semget()` es el valor clave (en nuestro caso devuelto por la llamada a `ftok()`). Este valor clave se compara con los valores clave existentes en el núcleo para otros conjuntos de semáforos. Ahora, las operaciones de apertura o acceso depende del contenido del argumento `semflg`.

IPC_CREAT

Crea el juego de semáforos si no existe ya en el núcleo.

IPC_EXCL

Al usarlo con IPC_CREAT, falla si el conjunto de semáforos existe ya.

Si se usa `IPC_CREAT` solo, `semget()`, bien devuelve el identificador del semáforo para un conjunto nuevo creado, o devuelve el identificador para un conjunto que existe con el mismo valor clave. Si se usa `IPC_EXCL` junto con `IPC_CREAT`, entonces o se crea un conjunto nuevo, o si el conjunto existe, la llamada falla con -1.`IPC_EXCL` es inútil por sí mismo, pero cuando se combina con `IPC_CREAT`, se puede usar como una facilidad garantizar que ningún semáforo existente se abra accidentalmente para accederlo.

Como sucede en otros puntos del IPC del Sistema V, puede aplicarse a los parámetros anteriores, un número octal para dar la máscara de permisos de acceso de los semáforos. Debe hacerse con una operación OR binaria.

El argumento `nsems` especifica el número de semáforos que se deben crear en un conjunto nuevo. Éste representa el número de impresores en nuestro cuarto de impresión ficticio descrito antes. El máximo número de semáforos en un conjunto se define en"linux/sem.h" como:

```
#define SEMMSL  32      /* <=512 max num de semaforos por id */
```

Observe que el argumento `nsems` se ignora si abre explícitamente un conjunto existente.

Creemos ahora una función de cobertura para abrir o cerrar juegos de semáforos:

```
int abrir_conj_semaforos( key_t clave, int numsems )
{
        int     sid;

        if ( ! numsems )
                return(-1);

        if((sid = semget( clave, numsems, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }

        return(sid);
}
```

Vea que se usan explícitamente los permisos `0660`. Esta pequeña función retornará, bien un identificador entero del conjunto de semáforos, o bien un -1 si hubo un error. En el ejemplo del final de esta sección, observe la utilización del flag IPC_EXCL para determinar si el conjunto de semáforos existe ya o no.

LLAMADA AL SISTEMA: semop()

```
  LLAMADA AL SISTEMA: semop();
  PROTOTIPO: int semop ( int semid, struct sembuf *sops, unsigned nsops);
    RETURNS: 0 si éxito (todas las operaciones realizadas)
             -1 si error: errno = E2BIG (nsops mayor que máx. número de opers.
                                  permitidas atómicamente)
                                  EACCESS (permiso denegado)
                                  EAGAIN (IPC_NOWAIT incluido, la operacion no terminó)
                                  EFAULT (dirección no válida en el parámetro sops)
                                  EIDRM (el conj. de semáforos fue borrado)
                                  EINTR (Recibida señal durante la espera)
                                  EINVAL (el conj. no existe, o semid inválido)
                                  ENOMEM (SEM_UNDO incluido, sin memoria suficiente para
                                          crear la estructura de retroceso necesaria)
                                  ERANGE (valor del semáforo fuera de rango)
  NOTAS:
```

El primer argumento de `semget()` es el valor clave (en nuestro caso devuelto por una llamada a `semget`). El segundo argumento (`sops`) es un puntero a un array de operaciones para que se ejecuta en el conjunto de semáforo, mientras el tercer argumento (`nsops`) es el número de operaciones en ese array.

El argumento `sops` apunta a un array del tipo `sembuf`. Se declara esta estructura en `linux/sem.h` como sigue:

```
/* La llamada al sistema semop usa un array de este tipo */
struct sembuf {
        ushort  sem_num;        /* posicion en el array */
        short   sem_op;         /* operacion del semaforo */
        short   sem_flg;        /* flags de la operacion */
};
```

`sem_num`

Número de semáforo sobre el que desea actuar

`sem_op`

Operación a realizar (positiva, negativa o cero)

`sem_flg`

Flags (parámetros) de la operación

Si `sem_op` es negativo, entonces su valor se resta del valor del semáforo. Éste pone en correlación con la obtención de recursos que controla el semáforo o los monitores de acceso. Si no se especifica `IPC_NOWAIT`, entonces proceso que efectúa la llamada duerme hasta que los recursos solicitados están disponible en el semáforo (otro proceso ha soltado algunos).

Si `sem_op` es positivo, entonces su valor se añade al semáforo. Éste se pone en correlación con los recursos devueltos al conjunto de semáforos de la aplicación. ¡Siempre se deben devolver los recursos al conjunto de semáforos cuando ya no se necesiten más!

Finalmente, si `sem_op` vale cero (0), entonces el proceso que efectúa la llamada dormirá hasta que el valor del semáforo sea 0. Éste pone en correlación la espera a un semáforo para obtener un 100ajusta dinámicamente el tamaño del conjunto de semáforos si obtiene utilización plena.

Para explicar la llamada de `semop`, volvamos a ver nuestro ejemplo de impresión. Supongamos una única una impresora, capaz de único un trabajo en un instante. Creamos un conjunto de semáforos con único semáforo en él (sólo una impresora), e inicializa ese semáforo con un valor de uno (único un trabajo en un instante).

Cada vez que deseemos enviarle un trabajo a esta impresora, primeros necesitamos asegura que el recurso está disponible. Hacemos este para intentar obtener *una unidad* del semáforo. Cargamos un array `sembuf` para realizar la operación:

```
struct sembuf sem_lock = { 0, -1, IPC_NOWAIT };
```

La traducción de la inicialización de la anterior estructura indica que un valor de "-1" se añadirá al número del semáforo 0 del conjunto de semáforos. En otras palabras, se obtendrá una unidad de recursos del único semáforo de nuestro conjunto (miembro 0). Se especifica **IPC_NOWAIT**, así la llamada o se produce inmediatamente, o falla si otro trabajo de impresión está activo en ese momento. Aquí hay un ejemplo de como usar esta inicialización de la estructura `sembuf` con la llamada al sistema `semop`:

```
if((semop(sid, &sem_lock, 1) == -1)
        perror("semop");
```

El tercer argumento (`nsops`) dice que estamos sólo ejecutando una (1) operación (hay sólo una estructura `sembuf` en nuestra array de operaciones). El argumento `sid` es el identificador IPC para nuestro conjunto de semáforos.

Cuando nuestro trabajo de impresión ha terminado, debemos *devolver* los recursos al conjunto de semáforos, de manera que otros puedan usar la impresora.

```
struct sembuf sem_unlock = { 0, 1, IPC_NOWAIT };
```

La traducción de la estructura anteriormente inicializada indica que un valor de "1" se agrega a semáforo número 0 en el conjunto de semáforos. En otras palabras, una unidad de recursos se devolverá al conjunto.

LLAMADA AL SISTEMA: semctl()

```
LLAMADA AL SISTEMA: semctl();
PROTOTIPO: int semctl ( int semid, int semnum, int cmd, union semun arg );
  RETURNS: entero positivo si éxito
           -1 si error: errno = EACCESS (permiso denegado)
                                EFAULT (dirección inválida en el argumento arg)
                                EIDRM (el juego de semáforos fue borrado)
                                EINVAL (el conj. no existe, o semid no es válido)
                                EPERM (El EUID no tiene privilegios para el comando
                                incluido en arg)
                                ERANGE (Valor para semáforo fuera de rango)
NOTAS: Realiza operaciones de control sobre conjuntos de semáforos
```

La llamada al sistema *semctl* se usa para desempeñar operaciones de control sobre un conjunto de semáforo. Esta llamada es análoga a la llamada al sistema *msgctl* que se usa para operaciones sobre las colas de mensaje. Si usted compara las listas de argumento de las dos llamadas al sistema, notará que la lista para *semctl* varía ligeramente con la de *msgctl*. La rellamada a semáforos actualmente implementada semáforos se implementa realmente conjuntos, más a entidades simples. Con las operaciones de semáforo operaciones, no sólo hace que se necesite pasar la clave IPC, sino que el semáforo destino dentro de el conjunto también.

Las llamados al sistema utilizan un argumento *cmd*, para la especificación del comando para ser realizado sobre el objeto IPC . La diferencia que resta está en el argumento final a ambas llamadas. En *msgctl*, el argumento final representa una copia de las estructuras de datos internos usado por el núcleo. Recuerde que nosotros usamos esta estructura para recuperar información interna sobre una cola de mensaje, así como también para colocar o cambiar permisos y propietario de la cola. Con semáforos, se soportan los comandos operacionales adicionales, así requieren unos tipos de estructuras de datos más complejos como el argumento final. El uso de un tipo *union* confunde muchos programadores novatos de los semáforo de forma considerable. Nosotros estudiaremos esta estructura cuidadosamente, en un esfuerzo para impedir cualquier confusión.

El argumento `cmd` representa el comando a ejecutar con el conjunto. Como puede ver, incluye los conocidos comandos IPC_STAT/IPC_SET, junto a otros específicos de conjuntos de semáforos:

IPC_STAT

Obtiene la estructura semid_ds de un conjunto y la guarda en la dirección del argumento `buf` en la unión `semun`.

IPC_SET

Establece el valor del miembro `ipc_perm` de la estructura semid_ds de un conjunto. Obtiene los valores del argumento `buf` de la unión `semun`.

IPC_RMID

Elimina el conjunto de semáforos.

GETALL

Se usa para obtener los valores de todos los semáforos del conjunto. Los valores enteros se almacenan en un array de enteros cortos sin signo, apuntado por el miembro *array* de la unión.

GETNCNT

Devuelve el número de procesos que esperan recursos.

GETPID

Retorna el PID del proceso que realizó la última llamada *semop*.

GETVAL

Devuelve el valor de uno de los semáforos del conjunto.

GETZCNT

Devuelve el número de procesos que esperan la disponibilidad del 100% de recurso.

SETALL

Coloca todos los valores de semáforos con una serie de valores contenidos en elmiembro *array* de la unión.

SETVAL

Coloca el valor de un semáforo individual con el miembro *val* de la unión.

El argumento `arg` representa un ejemplo de tipo `semun`. Esta unión particular se declara en `linux/sem.h` como se indica a continuación:

```
        /* argumento para llamadas a semctl */
        union semun {
                int val;                /* valor para SETVAL */
                struct semid_ds *buf;   /* buffer para IPC_STAT e IPC_SET */
                ushort *array;          /* array para GETALL y SETALL */
                struct seminfo *__buf;  /* buffer para IPC_INFO */
                void *__pad;
        };
```

`val`

Se usa con el comando SETVAL, para indicar el valor a poner en el semáforo.

`buf`

Se usa con los comandos IPC_STAT/IPC_SET. Es como una copia de la estructura de datos interna que tiene el núcleo para los semáforos.

`array`

Puntero que se usa en los comandos GETALL/SETALL. Debe apuntar a una matriz de números enteros donde se ponen o recuperan valores de los semáforos.

Los demás argumentos, *__buf* y *__pad*, se usan internamente en el núcleo y no son de excesiva utilidad para el programador. Además son específicos para el sistema operativo Linux y no se encuentran en otras versiones de UNIX.

Ya que esta llamada al sistema es de las más complicadas que hemos visto, pondremos diversos ejemplos para su uso.

La siguiente función devuelve el valor del semáforo indicado. El último argumento de la llamada (la unión), es ignorada con el comando GETVAL por lo que no la incluimos:

```
int obtener_sem_val( int sid, int semnum )
{
        return( semctl(sid, semnum, GETVAL, 0));
}
```

Volviendo al ejemplo de las impresoras, así obtendremos el estado de las cinco máquinas:

```
        #define MAX_IMPR 5

        uso_impresoras()
        {
                int x;

                for(x=0; x<MAX_IMPR; x++)
                        printf("Impresora %d: %d\n\r", x, obtener_sem_val( sid, x ));
        }
```

Considérese la siguiente función, que se debe usar para iniciar un nuevo semáforo:

```
void iniciar_semaforo( int sid, int semnum, int initval)
{
        union semun semopts;

        semopts.val = initval;
        semctl( sid, semnum, SETVAL, semopts);
}
```

Observe que el argumento final de *semctl* es una copia de la unión, más que un puntero a él. Mientras nosotros estamos en el tema de la unión como argumento, me permito demostrar una equivocación más bien común cuando usa este llamado de sistema.

Recordamos del proyecto msgtool que los comandos IPC_STAT e IPC_SET se usaron para alterar los permisos sobre la cola. Mientras estos comandos se soportan, en la implementación de un semáforo implementación, su uso es un poco diferente, como las estructuras de datos internas e recuperan y copian desde un miembro de la unión, más bien que una entidad simple. ¿ Puede encontrar el error en este código?

```
/* Los permisos se pasan como texto (ejemplo: "660") */

void changemode(int sid, char *mode)
{
        int rc;
        struct semid_ds mysemds;

        /* Obtener valores actuales */
        if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1)
        {
```

```
                perror("semctl");
                exit(1);
        }

        printf("Antiguos permisos: %o\n", semopts.buf->sem_perm.mode);

        /* Cambiar los permisos del semaforo */
        sscanf(mode, "%o", &semopts.buf->sem_perm.mode);

        /* Actualizar estructura de datos interna */
        semctl(sid, 0, IPC_SET, semopts);

        printf("Actualizado...\n");
}
```

El código intenta de hacer una copia local de las estructuras de datos internas estructura para el conjunto, modifica los permisos, e IPC_SET los devuelve al núcleo. Sin embargo, la primera llamada a *semctl* devuelve EFAULT, o dirección errónea para el último argumento (¡la unión!). Además, si no hubíeramos verificado los errores de la llamada, nosotros habríamos conseguido un fallo de memoria. ¿Por qué?

Recuerde que los comandos IPC_SET/IPC_STAT usan el miembro *buf* de la unión, que es un puntero al tipo *semid_ds*. ¡Los punteros, son punteros, son punteros y son punteros! El miembro *buf* debe indicar alguna ubicación válida de almacenamiento para que nuestra función trabaje adecuadamente. Considere esta versión:

```
void cambiamodo(int sid, char *mode)
{
        int rc;
        struct semid_ds mysemds;

        /* Obtener valores actuales de estructura interna */

        /* !Antes de nada apuntar a nuestra copia local! */
        semopts.buf = &mysemds;

        /* !Intentemos esto de nuevo! */
        if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1)
        {
                perror("semctl");
                exit(1);
        }

        printf("Antiguos permisos: %o\n", semopts.buf->sem_perm.mode);

        /* Cambiar permisos */
```

```
        sscanf(mode, "%o", &semopts.buf->sem_perm.mode);

        /* Actualizar estructura interna */
        semctl(sid, 0, IPC_SET, semopts);

        printf("Actualizado...\n");
}
```

---

semtool: Manipulador interactivo de semáforos

Vistazo Rápido El programa `semtool` usa la línea de comandos para determinar su comportamiento: es especialmente útil en los guiones de *shell*. Incluye todas las operaciones posibles para un conjunto de semáforos y puede usarse para controlar recursos compartidos desde los guiones de *shell*.

Sintaxis de la utilidad

Creación de un conjunto de semáforos

```
semtool c (número de semáforos en el conjunto)
```

Bloqueo de un semáforo

```
semtool b (número de semáforo a bloquear)
```

Desbloqueo de un semáforo

```
semtool d (número de semáforo a liberar)
```

Cambio de los permisos (modo)

```
semtool m (modo)
```

Borrado de un conjunto de semáforos

```
semtool b
```

Ejemplos

```
semtool  c 5
semtool  b
semtool  d
semtool  m 660
semtool  b
```

Código fuente

```
/*****************************************************************************
 Parte de la "Guia Linux de Programacion - Capitulo 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULO: semtool.c
 *****************************************************************************
 Utilidad de manejo de semaforos del sistema IPC SYSV

 *****************************************************************************/

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEM_RESOURCE_MAX        1       /* Valor inicial de todo semaforo */

void opensem(int *sid, key_t key);
void createsem(int *sid, key_t key, int members);
void locksem(int sid, int member);
void unlocksem(int sid, int member);
void removesem(int sid);
unsigned short get_member_count(int sid);
int getval(int sid, int member);
void dispval(int sid, int member);
void changemode(int sid, char *mode);
void usage(void);

int main(int argc, char *argv[])
{
        key_t key;
        int   semset_id;

        if(argc == 1)
                usage();

        /* Crear clave IPC unica */
        key = ftok(".", 's');

        switch(tolower(argv[1][0]))
        {
                case 'c': if(argc != 3)
```

```
                                        usage();
                                createsem(&semset_id, key,  atoi(argv[2]));
                                break;
                case 'b': if(argc != 3)
                                        usage();
                                opensem(&semset_id, key);
                                locksem(semset_id, atoi(argv[2]));
                                break;
                case 'd': if(argc != 3)
                                        usage();
                                opensem(&semset_id, key);
                                unlocksem(semset_id, atoi(argv[2]));
                                break;
                case 'b': opensem(&semset_id, key);
                                removesem(semset_id);
                                break;
                case 'm': opensem(&semset_id, key);
                                changemode(semset_id, argv[2]);
                                break;
                 default: usage();

        }

        return(0);
}

void opensem(int *sid, key_t key)
{
        /* Abrir (no crear!) el conjunto de semaforos */

        if((*sid = semget(key, 0, 0666)) == -1)
        {
                printf("No existe el conjunto de semaforos!\n");
                exit(1);
        }

}

void createsem(int *sid, key_t key, int members)
{
        int cntr;
        union semun semopts;

        if(members > SEMMSL) {
                printf("Lo siento: el numero maximo de semaforos es de: %d\n",
                        SEMMSL);
                exit(1);
```

```
        }

        printf("Intentando crear un conjunto de %d miembros\n",
                                members);

        if((*sid = semget(key, members, IPC_CREAT|IPC_EXCL|0666))
                        == -1)
        {
                fprintf(stderr, "El conjunto ya existe!\n");
                exit(1);
        }

        semopts.val = SEM_RESOURCE_MAX;

        /* Iniciar todos los miembros (puede hacerse con SETALL) */
        for(cntr=0; cntr<members; cntr++)
                semctl(*sid, cntr, SETVAL, semopts);
}

void locksem(int sid, int member)
{
        struct sembuf sem_lock={ 0, -1, IPC_NOWAIT};

        if( member<0 || member>(get_member_count(sid)-1))
        {
                fprintf(stderr, "miembro %d fuera de rango\n", member);
                return;
        }

        /* Intentamos bloquear el conjunto */
        if(!getval(sid, member))
        {
                fprintf(stderr, "Recursos del semaforo agotados (no bloqueo)!\n");
                exit(1);
        }

        sem_lock.sem_num = member;

        if((semop(sid, &sem_lock, 1)) == -1)
        {
                fprintf(stderr, "Fallo en bloqueo\n");
                exit(1);
        }
        else
                printf("Recursos decrementados en 1 (bloqueo)\n");

        dispval(sid, member);
```

```
}

void unlocksem(int sid, int member)
{
        struct sembuf sem_unlock={ member, 1, IPC_NOWAIT};
        int semval;

        if( member<0 || member>(get_member_count(sid)-1))
        {
                fprintf(stderr, "miembro %d fuera de rango\n", member);
                return;
        }

        /* Esta bloqueado? */
        semval = getval(sid, member);
        if(semval == SEM_RESOURCE_MAX) {
                fprintf(stderr, "Semaforo no bloqueado!\n");
                exit(1);
        }

        sem_unlock.sem_num = member;

        /* Intentamos desbloquear */
        if((semop(sid, &sem_unlock, 1)) == -1)
        {
                fprintf(stderr, "Fallo en desbloqueo\n");
                exit(1);
        }
        else
                printf("Recursos incrementados en 1 (desbloqueo)\n");

        dispval(sid, member);
}

void removesem(int sid)
{
        semctl(sid, 0, IPC_RMID, 0);
        printf("Semaforo borrado\n");
}

unsigned short get_member_count(int sid)
{
        union semun semopts;
        struct semid_ds mysemds;

        semopts.buf = &mysemds;
```

```
        /* Devolver numero de miembros */
        return(semopts.buf->sem_nsems);
}

int getval(int sid, int member)
{
        int semval;

        semval = semctl(sid, member, GETVAL, 0);
        return(semval);
}

void changemode(int sid, char *mode)
{
        int rc;
        union semun semopts;
        struct semid_ds mysemds;

        /* Obtener valores de la estructura interna */
        semopts.buf = &mysemds;

        rc = semctl(sid, 0, IPC_STAT, semopts);

        if (rc == -1) {
                perror("semctl");
                exit(1);
        }

        printf("Permisos antiguos: %o\n", semopts.buf->sem_perm.mode);

        /* Cambiar los permisos */
        sscanf(mode, "%ho", &semopts.buf->sem_perm.mode);

        /* Actualizar estructura interna */
        semctl(sid, 0, IPC_SET, semopts);

        printf("Actualizado...\n");

}

void dispval(int sid, int member)
{
        int semval;

        semval = semctl(sid, member, GETVAL, 0);
        printf("El semval del miembro %d es %d\n", member, semval);
}
```

```
void usage(void)
{
        fprintf(stderr, "semtool - Utilidad de manejo de semaforos\n");
        fprintf(stderr, "\nUSAGE:  semtool (c)rear <cuantos>\n");
        fprintf(stderr, "                 (b)loquear <sem #>\n");
        fprintf(stderr, "                 (d)esbloquear <sem #>\n");
        fprintf(stderr, "                 (b)orrar\n");
        fprintf(stderr, "                 (m)odo <modo>\n");
        exit(1);
}
```

semstat: utilidad para semtool

Como regalo final, incluimos el código fuente de una utilidad adicional llamada `semstat`. Este programa muestra los valores de cada semáforo del conjunto creado con `semtool`.

```
/*****************************************************************************
 Parte de la "Guia Linux de Programacion - Capitulo 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULO: semstat.c
 *****************************************************************************
 semstat muestra el estado de los semaforos manejados con semtool
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int get_sem_count(int sid);
void show_sem_usage(int sid);
int get_sem_count(int sid);
void dispval(int sid);

int main(int argc, char *argv[])
{
        key_t key;
        int   semset_id;

        /* Obtener clave IPC unica */
        key = ftok(".", 's');

        /* Abrir (no crear!) el conjunto de semaforos */
```

```
        if((semset_id = semget(key, 1, 0666)) == -1)
        {
                printf("El conjunto no existe\n");
                exit(1);
        }

        show_sem_usage(semset_id);
        return(0);
}

void show_sem_usage(int sid)
{
        int cntr=0, maxsems, semval;

        maxsems = get_sem_count(sid);

        while(cntr < maxsems) {
                semval = semctl(sid, cntr, GETVAL, 0);
                printf("Semaforo #%d:  --> %d\n", cntr, semval);
                cntr++;
        }
}

int get_sem_count(int sid)
{
        int rc;
        struct semid_ds mysemds;
        union semun semopts;

        /* Obtener valores de la estructura interna */
        semopts.buf = &mysemds;

        if((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1) {
                perror("semctl");
                exit(1);
        }

        /* devolver numero de semaforos del conjunto */
        return(semopts.buf->sem_nsems);
}

void dispval(int sid)
{
        int semval;

        semval = semctl(sid, 0, GETVAL, 0);
        printf("semval vale %d\n", semval);
```

```
}
```

6.4.4 Memoria Compartida

Conceptos Básicos

La memoria compartida se puede describir mejor como el plano (mapping) de un área (segmento) de memoria que se combinará y compartirá por más de un de proceso. Esta es por mucho la forma más rápida de IPC, porque no hay intermediación (es decir, un tubo, una cola de mensaje, etc). En su lugar, la información se combina directamente en un segmento de memoria, y en el espacio de direcciones del proceso llamante. Un segmento puede ser creado por un proceso, y consecutivamente escrito a y leído por cualquier número de procesos.

Estructuras de Datos Internas y de Usuario

Echemos un vistazo a las estructuras de datos que mantiene el núcleo para cada segmento de memoria compartida.

Estructura `shmid_ds` del Núcleo Como con la cola de mensaje y los conjuntos de semáforos, el núcleo mantiene unas estructuras de datos internas especiales para cada segmento compartido de memoria que existe dentro de su espacio de direcciones. Esta estructura es de tipo `shmid_ds`, y se define en `linux/shm.h` como se indica a continuación:

```
        /* Por cada segmento de memoria compartida, el nucleo mantiene
           una estructura como esta */
        struct shmid_ds {
                struct ipc_perm shm_perm;        /* permisos operacion */
                int     shm_segsz;               /* tamanyo segmento (bytes) */
                time_t  shm_atime;               /* instante ultimo enlace */
                time_t  shm_dtime;               /* instante ult. desenlace */
                time_t  shm_ctime;               /* instante ultimo cambio */
                unsigned short  shm_cpid;        /* pid del creador */
                unsigned short  shm_lpid;        /* pid del ultimo operador */
                short   shm_nattch;              /* num. de enlaces act. */

                                                 /* lo que sigue es privado */

                unsigned short   shm_npages;     /* tam. segmento (paginas) */
                unsigned long   *shm_pages;      /* array de ptr. a marcos -> SHMMAX */
                struct vm_area_struct *attaches; /* descriptor de enlaces */
        };
```

Las operaciones sobre esta estructura son realizadas por una llamada especial al sistema, y no deberían ser realizadas directamente. Aquí se describen de los campos más importantes:

`shm_perm`

Este es un ejemplo de la estructura `ipc_perm`, que se define en `linux/ipc.h`. Esto tiene la información de permisos para el segmento, incluyendo los permisos de acceso, e información sobre el creador del segmento (uid, etc).

`shm_segsz`

Tamaño del segmento (en bytes).

`shm_atime`

Instante del último enlace al segmento por parte de algún proceso.

`shm_dtime`

Instante del último desenlace del segmento por parte de algún proceso.

`shm_ctime`

Instante del último cambio de esta estructura (cambio de modo, etc).

`shm_cpid`

PID del proceso creador.

`shm_lpid`

PID del último proceso que actuó sobre el segmento.

`shm_nattch`

Número de procesos actualmente enlazados con el segmento.

LLAMADA AL SISTEMA: shmget()

Para crear un nuevo segmento de memoria compartida, o acceder a una existente, tenemos la llamada al sistema `shmget()`.

```
LLAMADA AL SISTEMA: shmget();

PROTOTIPO: int shmget ( key_t key, int size, int shmflg );
  RETORNA: si éxito, ident. de segmento de memoria compartida
           -1 si error: errno = EINVAL (Tam. de segmento invalido)
                                EEXIST (El segmento existe, no puede crearse)
                                EIDRM (Segmento borrado o marcado para borrarse)
                                ENOENT (No existe el segmento)
                                EACCES (Permiso denegado)
                                ENOMEM (No hay memoria suficiente)
NOTAS:
```

Esta llamada particular debería parecer casi como vieja conocida a estas alturas. Es parecido a las correspondientes para las colas de mensaje y conjuntos de semáforos.

El argumento primero de `shmget()` es el valor clave (en nuestro caso vuelto por una llamada a `ftok()`). Este valor clave se compara entonces a valores claves existentes que existen dentro de el núcleo para los otros segmentos compartidos de memoria. En esta situación, las operaciones de apertura o de acceso dependen de los contenidos del argumento `shmflg`.

IPC_CREAT

Crea un segmento si no existe ya en el núcleo.

IPC_EXCL

Al usarlo con IPC_CREAT, falla si el segmento ya existe.

Si se usa `IPC_CREAT` sin nada más, `shmget()` retornará, bien el identificador del segmento recién creado, o bien el de un segmento que existía ya con la misma clave IPC. Si se añade el comando `IPC_EXCL`, en caso de existir ya el segmento fallará, y si no se creará.

De nuevo, puede añadirse un modo de acceso en octal, mediante la operación OR.

Preparemos una función recubridora para crear o localizar segmentos de memoria compartida:

```
int abrir_segmento( key_t clave, int tamanyo )
{
        int     shmid;

        if((shmid = shmget( clave, tamanyo, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }

        return(shmid);
}
```

Observe el uso de los permisos explícitos `0660`. Esta sencilla función retornará un entero con el identificador del segmento, o -1 si hay error. Los argumentos son, el valor de la clave IPC y el tamaño deseado para el segmento (en bytes).

Una vez que un proceso obtiene un identificador de segmento válido, el siguiente paso es *mapear* (`attach`) el segmento en su propio espacio de direcciones.

LLAMADA AL SISTEMA: shmat()

```
  LLAMADA AL SISTEMA: shmat();

  PROTOTIPO: int shmat ( int shmid, char *shmaddr, int shmflg);
    RETORNA: dirección de acceso al segmento, o
```

```
           -1 si error: errno = EINVAL (Identificador o dirección inválidos)
                                ENOMEM (No hay memoria suficiente para ligarse)
                                EACCES (Permiso denegado)
  NOTAS:
```

Si el argumento `addr` es nulo (0), el núcleo intenta encontrar una zona no mapeada. Es la forma recomendada de hacerlo. Se puede incluir una dirección, pero es algo que solo suele usarse para facilitar el uso con hardware propietario o resolver conflictos con otras aplicaciones. El flag SHM_RND puede pasarse con un OR lógico en el argumento `shmflg` para forzar una dirección pasada para ser página (se redondea al tamaño más cercano de página).

Además, si se hace OR con el flag SHM_RDONLY y con el argumento de banderas, entonces el segmento compartido de memoria se mapea, pero marcado como sólo lectura.

Esta llamada es quizás la más simple de usar. Considere esta función de envoltura, que se pasa un identificador IPC válido para un segmento, y devuelve la dirección a la que el segmento está enlazado:

```
char *ligar_segmento( int shmid )
{
        return(shmat(shmid, 0, 0));
}
```

Una vez un segmento ha sido adecuadamente adjuntado, y un proceso tiene un puntero al comienzo del segmento, la lectura y la escritura en el segmento llegar a ser tan fácil como simplemente referenciar el puntero. ¡Tenga cuidado de no perder el valor del puntero original! Si esto sucede, no habrá ninguna manera de acceder a la base (comienzo) del segmento.

LLAMADA AL SISTEMA: shmctl()

```
  LLAMADA AL SISTEMA: shmctl();
  PROTOTYPE: int shmctl ( int shmqid, int cmd, struct shmid_ds *buf );
    RETURNS: 0 si éxito
             -1 si error: errno = EACCES (No hay permiso de lectura y cmd es IPC_STAT)
                                  EFAULT (Se ha suministrado una dirección inválida
                                  para los comandos IPC_SET e IPC_STAT)
                                  EIDRM  (El segmento fue borrado durante esta operación)
                                  EINVAL (shmqid inválido)
                                  EPERM  (Se ha intentado, sin permiso de escritura,
                                          el comando IPC_SET o IPC_RMID)
      NOTAS:
```

Esta llamada particular se usa tras la llamada *msgctl* solicitando colas de mensaje. En vista de este hecho, no se discutirá en detalle demasiado. Los que valores válidos de comando son:

IPC_STAT

Obtiene la estructura shmid_ds de un segmento y la almacena en la dirección del argumento buf.

IPC_SET

Ajusta el valor del miembro ipc_perm de la estructura, tomando el valor del argumento buf.

IPC_RMID

Marca un segmento para borrarse.

El comado IPC_RMID no quita realmente un segmento del núcleo. Más bien, *marca* el segmento para eliminación. La eliminación real del mismo ocurre cuando el último proceso actualmente adjunto al segmento termina su relación con él. Por supuesto, si ningún proceso está actualmente asociado al segmento, la eliminación es inmediata.

Para separar adecuadamente un segmento compartido de memoria, un proceso invoca la llamada al sistema `shmdt`.

LLAMADA AL SISTEMA: shmdt()

```
LLAMADA AL SISTEMA: shmdt();

PROTOTIPO: int shmdt ( char *shmaddr );
  RETURNS: -1 si error: errno = EINVAL (Dir. de enlace inválida)
```

Cuando un segmento compartido de memoria no se necesita más por un proceso, se debe separar con una llamado al sistema. Como mencionamos antes, esto no es lo mismo que eliminar un segmento desde el núcleo! Después de separar con éxito, el miembro shm_nattch de la estructura shmid_ds se decrementa en uno. Cuando este valor alcanza el cero (0), el núcleo quitará físicamente el segmento.

shmtool: Manipulador de segmentos de memoria compartida

Vistazo rápido Nuestro ejemplo final de objetos Sistema V IPC serán las `shmtool`, que es una herraminenta de línea de comando para crear, leer, escribir, y borrar segmentos compartidos de memoria. Una vez más, como en los ejemplos previos, el segmento se crea durante cualquier operación, si no existía anterioramente.

Sintaxis del comando

Escribir cadenas en el segmento

```
shmtool e "text"
```

Leer cadenas del segmento

```
shmtool l
```

Cambiar permisos (modo)

```
shmtool m (mode)
```

Borrado del segmento

```
shmtool b
```

Ejemplos

```
shmtool  e   prueba
shmtool  e   "Esto es una prueba"
shmtool  l
shmtool  b
shmtool  m   660
```

Código Fuente

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SEGSIZE 100

main(int argc, char *argv[])
{
        key_t key;
        int   shmid, cntr;
        char  *segptr;

        if(argc == 1)
                usage();

        /* Obtener clave IPC */
        key = ftok(".", 'S');

        /* Abrir (crear si es necesario) el segmento de memoria compartida */
        if((shmid = shmget(key, SEGSIZE, IPC_CREAT|IPC_EXCL|0666)) == -1)
        {
                printf("El segmento existe - abriendo como cliente\n");

                /* El segmento existe - abrir como cliente */
                if((shmid = shmget(key, SEGSIZE, 0)) == -1)
```

```
                {
                        perror("shmget");
                        exit(1);
                }
        }
        else
        {
                printf("Creando nuevo segmento\n");
        }

        /* Ligar el proceso actual al segmento */
        if((segptr = shmat(shmid, 0, 0)) == -1)
        {
                perror("shmat");
                exit(1);
        }

        switch(tolower(argv[1][0]))
        {
                case 'e': writeshm(shmid, segptr, argv[2]);
                          break;
                case 'l': readshm(shmid, segptr);
                          break;
                case 'b': removeshm(shmid);
                          break;
                case 'm': changemode(shmid, argv[2]);
                          break;
                 default: usage();

        }
}

writeshm(int shmid, char *segptr, char *text)
{
        strcpy(segptr, text);
        printf("Hecho...\n");
}

readshm(int shmid, char *segptr)
{
        printf("valor de segptr: %s\n", segptr);
}

removeshm(int shmid)
{
        shmctl(shmid, IPC_RMID, 0);
        printf("Segmento marcado para borrado\n");
```

```
}

changemode(int shmid, char *mode)
{
        struct shmid_ds myshmds;

        /* Obtener valor actual de la estructura de datos interna */
        shmctl(shmid, IPC_STAT, &myshmds);

        /* Mostrar antiguos permisos */
        printf("Antiguos permisos: %o\n", myshmds.shm_perm.mode);

        /* Convertir y cargar el modo */
        sscanf(mode, "%o", &myshmds.shm_perm.mode);

        /* Actualizar el modo */
        shmctl(shmid, IPC_SET, &myshmds);

        printf("Nuevos permisos : %o\n", myshmds.shm_perm.mode);
}

usage()
{
        fprintf(stderr, "shmtool - Utilidad para usar memoria compartida\n");
        fprintf(stderr, "\nUSAGE:  shmtool (e)scribir <texto>\n");
        fprintf(stderr, "                (l)eer\n");
        fprintf(stderr, "                (b)orrar\n");
        fprintf(stderr, "                (m)odo <modo en octal>\n");
        exit(1);
}
```

Capítulo 7

Programación del Sonido

Un PC tiene por lo menos un dispositivo de sonido: el altavoz interno. Pero, usted puede comprar también una tarjeta de sonido para insertar en su PC para disponer de un dispositivo de sonido más sofisticado. Mire Linux Sound User's Guide o Sound-HOWTO - HOWTO para comprobar las tarjetas soportadas.

7.1 Programación del altavoz interno

Lo crea o no, el altavoz de su PC es parte de la consola de Linux y por tanto un dispositivo de carácter. Por tanto, para manipularlo usaremos llamadas ioctl(). Para el altavoz interno tenemos dos comandos:

1. KDMKTONE

   Genera un tono durante un tiempo especificado.

   Ejemplo: `ioctl (fd, KDMKTONE,(long) argumento)`.

2. KIOCSOUND

   Genera un tono sin fin, o para otro que suena actualmente.

   Ejemplo: `ioctl(fd,KIOCSOUND,(int) tono)`.

El `argumento` consiste en un valor de tono en su parte baja y la duración en la parte alta. El valor de tono no es la frecuencia. El temporizador del PC,el 8254, se cronometra a 1.19 MHz y por tanto es 1190000/frecuencia. La duración se mide en ticks de cronómetro. Las dos llamadas a ioctl vuelven inmediatamente y por consiguiente puede producir pitidos largos sin bloquear el programa mientras.
El comando KDMKTONE debe usarse para producir señales de aviso ya que no tiene que preocuparse de parar el tono.
El comando KIOCSOUND puede usarse para tocar canciones tal como se demuestra en el programa de ejemplo `splay` *(por favor, envíeme más ficheros* `.sng`*)*. Para parar el tono hay que usar el valor 0 en el `tono`.

7.2 Programación de una Tarjeta de sonido

Como programador, le resultará importante saber si el sistema sobre el que actúa tiene una tarjeta de sonido conectada. Para ello puede intentar abrir `/dev/sndstat`. Si falla y sale ENODEV en el valor de `errno`, es porque no hay ningún manejador de sonido activo. El mismo resultado puede obtenerse chequeando `/dev/dsp`, siempre que no sea un enlace al manejador `pcsnd` en cuyo caso la llamada `open()` no fallaría.

Si quiere intentarlo a nivel de hardware, deberá conocer alguna combinación de llamadas `outb()` e `inb()` para detectar la tarjeta que está buscando.

Utilizando el manejador de sonido en los programas, tiene la ventaja de que funcionará igual en otros sistemas 386, ya que la gente inteligente decidirá usar el mismo controlador en Linux, isc, FreeBSD y otras versiones de Unix de 386. Esto ayudará a transportar programas con sonido entre Linux y otras arquitecturas. Una tarjeta de sonido no es parte de la consola Linux, sino un dispositivo especial. En general ofrecerá las siguientes prestaciones:

- Muestreo digital, en entrada y salida
- Modulación de frecuencia, en salida
- Interfaz MIDI

Cada una de estas características tienen su propia interfaz con el controlador. Para el muestreo digital se usa `/dev/dsp`. Para la modulación de frecuencia se usa `/dev/sequencer`, y para la interfaz MIDI se usa `/dev/midi`. Los ajustes de sonido (tal como volumen o bajos), pueden controlarse mediante la interfaz `/dev/mixer`. Por compatibilidad se incluye también un dispositivo `/dev/audio`, capaz que reproducir datos de sonido SUN μ-law, que los mapea al dispositivo de muestreo digital.

Si supuso la utilización de `ioctl()` para manipular dispositivos de sonido, está en lo cierto. Las peticiones de esta clase se definen en $< linux/soundcard.h >$ y comienzan con SNDCTL_.

Puesto que no tengo una tarjeta de sonido, alguien con más conocimientos debería continuar este capítulo

Sven van der Meer v0.3.3, 19 Jan 1995

Capítulo 8

Gráficos en modo carácter

Este capítulo se dedica a las entradas y salidas de pantalla que no se basan en pixels, sino en caracteres. Cuando decimos carácter, queremos decir una composición de pixels que pueden cambiarse dependiendo de un conjunto de caracteres. Su tarjeta gráfica ya dispone de uno o más charsets y opera en modo texto por defecto, porque el texto puede procesarse mucho más rápido que un gráfico de pixel. Se pueden hacer más cosas con los terminales que simplemente mostrar texto. Describiré como usar los aspectos especiales que su terminal linux , especialmente los que ofrece la consola el linux.

- **printf, sprintf, fprintf, scanf, sscanf, fscanf**
  Con estas funciones de *libc* podrá realizar salida con formato sobre *stdout* (la salida estándar), *stderr* (la salida de errores estándar) y otros *streams* definidos como `FILE *stream` (ficheros, por ejemplo). La función **scanf(...)** proporciona un mecanismo similar para entradas con formato, desde *stdin* (la entrada estándar).

- **termcap**
  La base de datos *termcap* (CAPacidades de TERMinal) es un conjunto de entradas de descripción de terminales en el archivo ASCII /etc/termcap. Aquí puede encontrar la información sobre cómo mostrar caracteres especiales, como realizar operaciones (borrar, insertar caracteres o líneas, etc) y como inicializar un terminal. La base de datos se usa, por ejemplo, por el editor vi. Hay funciones de biblioteca de vista para leer y usar las capacidades terminales (*termcap(3x)*). Con esta base de datos, los programas pueden trabajar con una variedad de terminales con el mismo código. El uso de la biblioteca de funciones termcap y la base de datos proporciona sólo acceso a bajo nivel al terminal. El cambio de los atributos o colores o atributos, y la optimización debe ser hecha por el mismo programador.

- **base de datos terminfo**
  La base de datos *terminfo* (INFOrmación de TERMinales) se basa en la base de datos *termcap* y también describe las capacidades de las terminales, pero sobre un nivel más alto que termcap. Usando terminfo, el programa puede cambiar fácilmente los atributos, usar teclas especiales

tales como teclas de función y más. La base de datos puede encontrarse en */usr/lib/terminfo/[A-z,0-9]**. Cada archivo describe un de terminal.

- **curses**
  Terminfo es una base buena para usar en el manejo de un terminal en un programa. La biblioteca (BSD -)CURSES da acceso a alto nivel al terminal y se base en la base de datos terminfo. Curses le permite abrir y manipular ventanas sobre la pantalla, proporciona un conjunto completo de funciones de entrada y salida, y puede alterar atributos de video de una manera independiente del terminal sobre más de 150 terminales. La biblioteca de curses puede encontrarse en */usr/lib/libcurses.a*. Esta es el la versión BSD curses.

- **ncurses**
  Ncurses es la siguiente mejora. La versión 1.8.6debe ser compatible con curses de AT&T como se define en SYSVR4 y tiene algunas extensiones tales como manipulación de color, optimización especial para el salida, optimizaciones específicas de terminales, y más. Se ha probado sobre muchos sistemas tales como SUN-OS, HP y Linux. Yo recomiendo usar ncurses en vez de las otras. Sobre Unix SYSV de sistemas (tal como Sun Solaris) deber existir una biblioteca de curses con la misma funcionalidad que ncurses (realmente las curses de solaris tienen algunas funciones más y soporte de ratón).

En las secciones siguientes describiré como usar los diferentes paquetes diferentes para acceder a un terminal. Con Linux tenemos la versión GNU de termcap y nosotros podemos usar ncurses en vez de curses.

8.1 Funciones E/S en la libc

8.1.1 Salida con Formato

Las funciones del grupo **printf(...)** proporciona una salida formateada y permite la conversión de los argumentos.

- `int fprintf(FILE *stream, const char *formato, ...)`,
  transformará la salida (argumentos para rellenar en `...`) y lo escribirá en un stream. El formato definido en `formato` se escribe también. La función devuelve el número de caracteres escritos o un valor negativo en caso de error.

  El `formato` contiene dos tipos de objetos:

  1. caracteres normales para la salida
  2. información de cómo transformar o dar formato a los argumentos

  La información del formato debe comenzar con el símbolo %, seguido de valores apropiados para el formato y de un carácter para la traducción (para imprimir el propio signo % usaremos el comando `%%`). Los posibles valores para el formato son:

- Flags
  - `-`
    El argumento formateado se imprimirá en el margen izquierdo (por defecto va en el margen derecho del campo para el argumento).
  - `+`
    Cada número será impreso con su signo, por ejemplo `+12` o `-2.32`.
- Blanco
  Cuando el primer carácter no es un signo, se insertará un blanco.
- `0`
  Para transformaciones numéricas la anchura del campo se rellenará con ceros a la izquierda.
- `#`
  Salida alternativa dependiendo de las transformaciones para los argumentos
  - Para *o* el primer número es un *0*.
  - Para *x* o *X* se imprimirá *0x* o *0X* delante del argumento.
  - Para *e*, *E*, *f* o *F* la salida tiene punto decimal.
  - Para *g* o *G* se imprimen ceros al final del argumento.
- Un número para la amplitud mínima del campo.
  El argumento transformado se imprime en un campo que, al menos, es tan grande como el mismo argumento. Para números, puede hacer la anchura del campo más grande. Si el argumento formateado es más pequeño, entonces la anchura del campo se rellenará con ceros o blancos.
- Un punto separa la anchura del campo de la precisión.
- Un número para la precisión.

Valores posibles para la transformación están en la tabla 8.1 en la página 84.

- `int printf(const char *formato, ...)`
  Similar a **fprintf(stdout, ...)**.

- `int sprintf(char *s, const char *formato, ...)`
  Similar a **printf(...)**, con la salvedad de que la salida es escrita en la cadena apuntada por el puntero `s` (terminada en *0*).

  (**Nota:** Debe haberse reservado memoria suficiente para `s`.)

- `vprintf(const char *formato, va_list arg)`
  `vfprintf(FILE *stream, const char *formato, va_list arg)`
  `vsprintf(char *s, const char *formato, va_list arg)`
  Funciones similares a las anteriores, aunque ahora la lista de argumentos se introduce en `arg`.

Tabla 8.1: Libc - transformaciones en printf

Carácter	**Formateado a**
d,i	*entero* con signo, decimal
o	*entero* sin signo, octal, sin ceros a la izquierda
x,X	*entero* sin signo, hexadecimal sin cabecera 0x
u	*entero* sin signo, decimal
c	*entero* (sin signo), como carácter
s	*char* * hasta el \0
f	coma flotante (*double*), como [-]mmm.ddd
e,E	coma flotante (*double*) como [-]m.dddddde±xx
g,G	coma flotante (*double*) usando %e o %f si es necesario
p	*void* *
n	*int* *
%	%

8.1.2 Entrada con Formato

Igual que usamos **printf(...)** para salidas con formato, también podemos usar **scanf(...)** para entradas con formato.

- `int fscanf(FILE *stream, const char *formato, ...)`

  fscanf(...) lee de un `stream` y transformará la entrada con las reglas definidas en el `formato`. El resultado se sitúa en el argumento dado por .... (Observe que los argumentos deben ser punteros). La lectura termina cuando no hay más reglas de transformación en el formato. **fscanf(...)** devolverá EOF cuando la primera transformación alcance el final del archivo u ocurra algún error. En otro caso devolverá el número de argumentos transformados.

  El `formato` puede incluir reglas para dar formato a los caracteres de entrada (vea la tabla 8.2 en la página 85). También puede incluir: can include rules on how to format the input arguments

  – Espacios o tabuladores, que son ignorados.

  – Cualquier carácter normal (salvo %). Los caracteres deben estar en la entrada, en la misma posición.

  – Reglas de transformación, precedidas de un %, con el carácter opcional * (esto permitirá a **fscanf(...)** asignarlo a un argumento), un número opcional, un carácter opcional *h, l o L* (para la longitud del objetivo) y el carácter de transformación.

- `int scanf(const char *formato, ...)`
  Equivalente a **fscanf(stdin,...)**.

Tabla 8.2: Libc - transformaciones en scanf

Carácter	**Entrada - Tipo del argumento**
d	entero decimal - *int **
i	entero - *int ** (la entrada puede ser octal o hexadecimal)
o	entero octal - *int ** (con 0 a la izquierda opcional)
u	decimal, sin signo - *unsigned int **
x	entero hexadecimal - *int ** (con 0x a la izquierda opcional)
c	uno o más caracteres - *char ** (sin el \0)
s	caracteres (sin separadores) - *char ** (con el \0)
e,f,gf	coma flotante - *float ** (ej: [-]m.dddddde±xx)
p	puntero - *void **
n	número de argumentos transformados - *int **
[...]	caracteres de la entrada - *char **
[^...]	excluir esos caracteres - *char **
%	%

h puede ir antes de d,i,n,o,u y x, cuando el puntero es tipo *short*
l puede ir antes de d,i,n,o,u y x, cuando el puntero es *long*
l puede ir antes de e,f y g cuando el puntero es *double*
L puede ir antes de e,f y g cuando el puntero es *long double*

- `int sscanf(char *str, const char *format, ...)`
  Similar a **scanf(...)**, aunque ahora la entrada se lee de la cadena `str`.

8.2 La Librería Termcap

8.2.1 Introducción

La librería Termcap es una API (Interfaz de Programación de Aplicación) con la base de datos *termcap* que se encuentra en */etc/termcap/*. Las funciones de esta librería permiten realizar las siguientes acciones:

- Obtener descripción del terminal actual: **tgetent(...)**.
- Buscar la descripción para información: **tgetnum(...)**, **tgetflag(...)**, **tgetstr(...)**.
- Codificar parámetros numéricos en la forma de un terminal específico: **tparam(...)**, **tgoto(...)**.
- Calcular y realizar rellenos **tputs(...)**.

Los progamas que usan la biblioteca termcap deben incluir *termcap.h* y deben ser enlazados con `libtermcap` de esta forma:

```
gcc [opciones] ficheros -ltermcap
```

Las funciones termcap son rutinas independientes del terminal, pero sólo dan al programador acceso a bajo nivel. Para un manejo de alto nivel tenemos que usar curses o ncurses.

8.2.2 Encontrar la descripción del terminal

- `int tgetent(void *buffer, const char *tipoterm)`
  En el sistema operativo Linux, el nombre de la clase de terminal actual se encuentra en la variable de entorno *TERM*. Por tanto, el argumento `tipoterm` lo obtendremos mediante la función *getenv(3)*.

  Cuando usamos la versión *termcap* de GNU (lo habitual bajo Linux), no es necesario reservar memoria para el `buffer`. En otras implementaciones habrá que reservar 2048 bytes (realmente son 1024, pero el tamaño fue doblado).

  tgetent(...) devuelve 1 cuando hay éxito, y 0 si no se encuentra información para ese terminal en la base de datos. Otros errores devolverán diferentes valores.

  El siguiente ejemplo nos ayudará a ver cómo se usa la función **tgetent(...)**:

```
#define buffer 0
char *tipoterm=getenv("TERM");
int ok;

ok=tgetent(buffer,tipoterm);
if(ok==1)
  /* todo va bien, se ha encontrado la informacion */
else if(ok==0)
 /* algo va mal con TERM
  * comprobar tipoterm y luego la propia base de datos
  */
else
 /* este caso corresponde a un error fatal */
```

  Por defecto, la base de datos se encuentra en */etc/termcap/*. Si existe la variable de entorno TERMCAP, por ejemplo con el valor *$HOME/mytermcap*, las funciones de la librería buscarán la base de datos en ese nuevo directorio. Sin barras iniciales en TERMCAP, el valor definido se usará como nombre y descripción del terminal.

8.2.3 Lectura de una descripción de terminal

Cada parte de la información se llama *capacidad*, cada capacidad, es un código de dos letras, y cada código de dos letras viene seguido de por el valor de la capacidad. Los tipos posibles son:

- **Numérico**: Por ejemplo, *co* – número de columnas
- **Booleano** o **Flag**: Por ejemplo, *hc* – terminal hardcopy

- **Cadena**: Por ejemplo, *st* – set tab stop

Cada capacidad está asociada con un valor individual. (*co* es siempre numérico, *hc* es siempre un flag y *st* es siempre un string).Hay tres tipos diferentes de valores, y por tanto hay tres funciones para interrogarlos. `char *nombre` es el código de dos letras para la capacidad.

- `int tgetnum(char *nombre)`
  Obtiene el valor de una capacidad que es numérica, tal como *co*. **tgetnum(...)** devuelve el valor numérico si la capacidad está disponible, en otro caso 1. (Observe que el valor devuelto no es negativo).

- `int tgetflag(char *nombre)`
  Obtiene el valor de una capacidad que es boolean (o flag). Devuelve 1 si la badera (flag) está presente, 0 en otro caso.

- `char *tgetstr(char *nombre, char **area)`
  Obtiene el valor de una capacidad que es un string. Devuelve un puntero al string o NULL si no está presente. En la versión GNU, si `area` es NULL, termcap ubicará memoria para él. Termcap no hará más referencia a ese puntero, de forma que no olvide liberar el `nombre` antes de terminar el programa. Este método es preferido, porque no tiene que saber cuánto espacio se necesita para el puntero, así que deje a termcap hacerlo por vd.

```
char *clstr, *cmstr;
int   lines,  cols;

void term_caps()
{
char *tmp;

 clstr=tgetstr("cl",0); /* limpiar pantalla */
 cmstr=tgetstr("cm",0); /* mover y,x     */

 lines=tgetnum("li"); /* lineas del terminal   */
 cols=tgetnum("co");  /* columnas del terminal */

 tmp=tgetstr("pc",0); /* caracter separador */

 PC=tmp ? *tmp : 0;
 BC=tgetstr("le",0); /* cursor un caracter a la izquierda */
 UP=tgetstr("up",0); /* cursor arriba una linea   */
}
```

8.2.4 Capacidades de Termcap

Capacidades Booleanas

`5i` La impresora no hará eco en la pantalla
`am` Márgenes automáticos
`bs` Control-H (8 dec.) realiza un backspace

`bw`	Backspace al margen izquierdo vuelve la margen derecho e la línea superior
`da`	Display retenido sobre la pantalla
`db`	Display retenido bajo la pantalla
`eo`	Un espacio borra todos los caracteres de la posición del cursor
`es`	Secuencias de Escape y caracteres especiales funcionan en la línea de estado
`gn`	Dispositivo Genérico
`hc`	Esto es un terminal de copia física (hardcopy terminal)
`HC`	El cursor es difícil de ver cuando no está en la línea de abajo
`hs`	Tiene línea de estado
`hz`	"Hazel tine bug", el terminal no puede imprimir caracteres con tilde
`in`	Terminal inserta nulos, no espacios, para rellenar blancos
`km`	Terminal tiene tecla "meta" (alt)
`mi`	Los movimientos del cursor funcionan en modo inserción
`ms`	Los movimientos del cursor funcionan en modo standout/subrayado
`NP`	Sin carácter de "padding"
`NR`	"ti" no invierte "te"
`nx`	No hay "padding", debe usarse XON/XOFF
`os`	Terminal can overstrike
`ul`	Terminal underlines although it can not overstrike
`xb`	f1 enví ESCAPE, f2 enví ˆC
`xn`	Newline/wraparound glitch
`xo`	El terminal usa protocolo xon/xoff
`xs`	Text typed over standout text will be displayed in standout
`xt`	Teleray glitch, destructive tabs and odd standout mode

Capacidades Numéricas

`co`	Número de columnas
`dB`	Retardo (ms) para el retroceso en terminales de copia física
`dC`	Retardo (ms) para el retorno de carro en terminales de copia física
`dF`	Retardo (ms) para alim. página en terminales de copia física
`dN`	Retardo (ms) para fin de línea en terminales de copia física
`dT`	Retardo (ms) para parada de tabulación en terminales de copia física
`dV`	Retardo (ms) para parada de tabulación vertical en terminales de copia física
`it`	Diferencia entre posiciones de tabulación
`lh`	Alto de los 'soft labels'
`lm`	Líneas de memoria
`lw`	Ancho de los 'soft labels'
`li`	Número de líneas
`Nl`	Número de 'soft labels'
`pb`	Mínimo ratio en baudios que necesita 'padding'
`sg`	Standout glitch
`ug`	Underline glitch
`vt`	Número de terminal virtual
`ws`	Ancho de línea de estado si difiere del ancho de pantalla

Capacidades con Cadenas

`!1`	tecla de salvar con shift
`!2`	tecla de suspensión con shift
`!3`	tecla de deshacer con shift
`#1`	tecla de ayuda con shift
`#2`	tecla de inicio con shift
`#3`	tecla de entrada con shift
`#4`	tecla con shift de cursor izquierda
`%0`	tecla de rehacer
`%1`	tecla de ayuda
`%2`	tecla de selección
`%3`	tecla de mensaje
`%4`	tecla de mover
`%5`	tecla de siguiente objeto
`%6`	tecla de abrir

`%7`	tecla de opciones
`%8`	tecla de objeto anterior
`%9`	tecla de imprimir
`%a`	tecla de mensajes con shift
`%b`	tecla de mover con shift
`%c`	tecla de siguiente, con shift
`%d`	tecla de opciones con shift
`%e`	tecla de anterior, con shift
`%f`	tecla de imprimir, con shift
`%g`	tecla de rehacer, con shift
`%h`	tecla de reemplazar, con shift
`%i`	tecla de cursor dcha. con shift
`%j`	tecla continuar con shift
`&0`	tecla cancelar con shift
`&1`	tecla de referencia
`&2`	tecla de refrescar
`&3`	tecla de reemplazar
`&4`	tecla de reinicio
`&5`	tecla de continuar
`&6`	tecla de salvar
`&7`	tecla de suspensión
`&8`	tecla deshacer
`&9`	tecla de inicio con shift
`*0`	tecla de buscar con shift
`*1`	tecla de comando con shift
`*2`	tecla de copiar con shift
`*3`	tecla de crear con shift
`*4`	carácter de borrado con shift
`*5`	borrar línea con shift
`*6`	tecla de selección
`*7`	tecla de fin con shift
`*8`	limpiar línea con shift
`*9`	tecla de salida con shift
`0`	tecla de buscar
`1`	tecla de inicio
`2`	tecla de cancelar
`3`	tecla de cerrar
`4`	tecla de comando
`5`	tecla de copiar
`6`	tecla de crear
`7`	tecla de fin
`8`	tecla de entrar/enviar
`9`	tecla de salir
`al`	Insertar una línea
`AL`	Insertar %1 líneas
`ac`	Pairs of block graphic characters to map alternate character set
`ae`	Fin de juego de caracteres alternativo
`as`	Iniciar juego de caracteres alternativo para caracteres grf́icos de bloques
`bc`	Carćter de retroceso, si no es ˆH
`bl`	Pitido acústico
`bt`	Moverse a la parada de tabulación previa

`cb`	Limpiar desde comienzo de línea hasta el cursor
`cc`	Carćter comodín de comando
`cd`	Limpiar hasta final de pantalla
`ce`	Limpiar hasta final de línea
`ch`	Mover cursor horizontalmente hasta la columna %1
`cl`	Limpiar pantalla y devolver cursor al principio
`cm`	Mover cursor a la fila %1 y columna %2 (de la pantalla)
`CM`	Mover cursor a la fila %1 y la columna %2 (de la memoria)
`cr`	Retorno de carro
`cs`	Mover región de línea %1 a la %2
`ct`	Limpiar tabuladores
`cv`	Mover cursor a la línea %1
`dc`	Borrar un carácter
`DC`	Borrar %1 caracteres
`dl`	Borrar una línea
`DL`	Borrar %1 líneas
`dm`	Inicio del modo borrado
`do`	Bajar cursor una línea
`DO`	Bajar cursor #1 líneas
`ds`	Desactivar línea de estado
`eA`	Activar juego de caracteres alternativo
`ec`	Borrar %1 caracteres desde el cursor
`ed`	Fin del modo borrado
`ei`	Fin del modo inserción
`ff`	Carácter de salto de pǵina en terminales de copia física
`fs`	Devolver carácter a posicion antes de ir a la línea de estado
`F1`	Cadena enviada por tecla f11
`F2`	Cadena enviada por tecla f12
`F3`	Cadena enviada por tecla f13
	... ...
`F9`	Cadena enviada por tecla f19
`FA`	Cadena enviada por tecla f20
`FB`	Cadena enviada por tecla f21
	... ...
`FZ`	Cadena enviada por tecla f45
`Fa`	Cadena enviada por tecla f46
`Fb`	Cadena enviada por tecla f47
	... ...
`Fr`	Cadena enviada por tecla f63
`hd`	Bajar el cursor una línea
`ho`	Vuelta del cursor al principio
`hu`	Mover cursor media línea arriba
`i1`	Cadena de inicio 1 al entrar (login)
`i3`	Cadena de inicio 2 al entrar (login)
`is`	Cadena de inicio 3 al entrar (login)
`ic`	Inserar un carácter

IC	Insertar %1 caracteres
if	Fichero de inicio
im	Entrar en modo inserción
ip	Insert pad time and needed special characters after insert
iP	Programa de inicio
K1	tecla superior izquierda en teclado de números
K2	tecla central en teclado de números
K3	tecla superior derecha en teclado de números
K4	tecla inferior izquierda en teclado de números
K5	tecla inferior derecha en teclado de números
k0	Tecla de función 0
k1	Tecla de función 1
k2	Tecla de función 2
k3	Tecla de función 3
k4	Tecla de función 4
k5	Tecla de función 5
k6	Tecla de función 6
k7	Tecla de función 7
k8	Tecla de función 8
k9	Tecla de función 9
k;	Tecla de función 10
ka	Limpiar todas las tabulaciones
kA	Tecla de insertar línea
kb	Tecla de retroceso
kB	Fin de tab. retroceso
kC	Tecla de limpiar pantalla
kd	Tecla de bajar cursor
kD	Tecla de borrar carácter en el cursor
ke	desactivar teclado de números
kE	Tecla de borrar hasta fin de línea
kF	Tecla de scroll adelante/abajo
kh	Tecla de regreso del cursor al inicio
kH	Cursor home down key
kI	Tecla de insertar carácter/modo de inserción
kl	Tecla de cursor izquierda
kL	Tecla de borrar línea
kM	Tecla de salir modo inserción
kN	Tecla de siguiente página
kP	Tecla de página anterior
kr	Tecla de cursor derecha
kR	Tecla de scroll atrś/arriba
ks	Activar teclado de números
kS	Tecla de borrar hasta fin de pantalla
kt	Tecla de limpiar esta tabulación
kT	Tecla para poner tab. aquí
ku	Tecla de cursor arriba
l0	Label of zeroth function key, if not f0
l1	Label of first function key, if not f1
l2	Label of first function key, if not f2
...	...
la	Label of tenth function key, if not f10
le	Cursor left one character
ll	Move cursor to lower left corner
LE	Cursor left %1 characters
LF	Turn soft labels off
LO	Turn soft labels on
mb	Start blinking
MC	Clear soft margins
md	Start bold mode
me	End all mode like so, us, mb, md and mr
mh	Start half bright mode
mk	Dark mode (Characters invisible)
ML	Set left soft margin
mm	Put terminal in meta mode
mo	Put terminal out of meta mode
mp	Turn on protected attribute
mr	Start reverse mode
MR	Set right soft margin
nd	Cursor right one character
nw	Carriage return command
pc	Padding character
pf	Turn printer off
pk	Program key %1 to send string %2 as if typed by user
pl	Program key %1 to execute string %2 in local mode
pn	Program soft label %1 to to show string %2
po	Turn the printer on
pO	Turn the printer on for %1 (<256) bytes
ps	Print screen contents on printer
px	Program key %1 to send string %2 to computer
r1	Reset string 1, set sane modes
r2	Reset string 2, set sane modes
r3	Reset string 3, set sane modes
RA	disable automatic margins
rc	Restore saved cursor position
rf	Reset string file name
RF	Request for input from terminal
RI	Cursor right %1 characters
rp	Repeat character %1 for %2 times
rP	Padding after character sent in replace mode
rs	Reset string
RX	Turn off XON/XOFF flow control
sa	Set %1 %2 %3 %4 %5 %6 %7 %8 %9 attributes

`SA` enable automatic margins
`sc` Save cursor position
`se` End standout mode
`sf` Normal scroll one line
`SF` Normal scroll %1 lines
`so` Start standout mode
`sr` Reverse scroll
`SR` scroll back %1 lines
`st` Set tabulator stop in all rows at current column
`SX` Turn on XON/XOFF flow control
`ta` move to next hardware tab
`tc` Read in terminal description from another entry
`te` End program that uses cursor motion
`ti` Begin program that uses cursor motion
`ts` Move cursor to column %1 of status line
`uc` Underline character under cursor and move cursor right
`ue` End underlining
`up` Cursor up one line
`UP` Cursor up %1 lines
`us` Start underlining
`vb` Visible bell
`ve` Normal cursor visible
`vi` Cursor invisible
`vs` Standout cursor
`wi` Set window from line %1 to %2 and column %3 to %4
`XF` XOFF character if not ^S

8.3 Ncurses - Introducción

Se usará la siguiente terminología a lo largo de este capítulo:

- ventana (*window*) - es una representación interna que contiene una imagen de una parte de la pantalla. WINDOW se define en *ncurses.h*.
- pantalla (*screen*) - es una ventana con el tamaño de toda la pantalla (desde el superior izquierdo al inferior derecho). *Stdscr* y *curscr* son pantallas.
- terminal - es una pantalla especial con información sobre lo que aparece en la pantalla actual.
- variables - Las siguientes son variables y constantes definidas en *ncurses.h*
  - WINDOW *curscr - pantalla actual
  - WINDOW *stdscr - pantalla estándar
  - int LINES - líneas del terminal
  - int COLS - columnas del terminal
  - bool TRUE - flag verdadero, 1
  - bool FALSE - flag falso, 0
  - int ERR - flag de error, -1
  - int OK - flag de corrección, 0
- funciones - los argumentos que llevan son de los siguientes tipos:
  - win - WINDOW*
  - bf - bool
  - ch - chtype
  - str - char*
  - chstr - chtype*
  - fmt - char*
  - en otro caso, int (entero)

Normalmente un programa que usa la biblioteca ncurses se parece a esto:

```
#include <ncurses.h>
...
main()
{
  ...
  initscr();
  /* Llamadas a funciones de ncurses */
  endwin();
  ...
}
```

La inclusión de *ncurses.h* definirá variables y tipos para ncurses, tales como WINDOW y prototipos de funciones. Incluye automaticamente *stdio.h*, *stdarg.h*, *termios.h* y *unctrl.h*.

La función **initscr()** se usa para inicializar las estructuras de datos ncurses y para leer el archivo terminfo apropiado. La memoria se reserva. Si ocurre un error, **initscr** devolverá ERR, en otro caso devuelve un puntero. Adicionalmente la pantalla se borra e inicializa.

La función **endwin()** libera todos los recursos para ncurses y restaura los modos de terminal al estado que tenían antes de llamar a **initscr()**. Se debe llamar antes de cualquier otra función de la biblioteca ncurses y **endwin()** debe ser llamado antes de que su programa termine. Cuando quiere salidas por más de un terminal, puede usar **newterm(...)** en lugar de **initscr()**.

Compílese el programa con:

```
gcc [opciones] ficheros -lncurses
```

En las opciones se incluirá cualquiera que precise (*ver gcc(1)*). Ya que el camino a ncurses.h ha cambiado, debe poner al menos la siguiente opción:

```
-I/usr/include/ncurses
```

En otro caso, no se encontrarán ni ncurses.h, nterm.h, termcap.h o unctrl.h. Otras posibles opciones en Linux son:

```
-O2 -ansi -Wall -m486
```

O2 pide a `gcc` cierta optimización, *-ansi* es para que compile código compatible con ANSI-C, *-Wall* imprimirá toda clase de avisos y *-m486* generará código optimizado para Intel 486 (aunque el código podrá correr también en un 386).

La librería ncurses está en /usr/lib/. Hay tres versiones de ésta:

- **libncurses.a** es la librería normal.
- **libdcurses.a** es la librería que permite depuración.
- **libpcurses.a** para análisis de perfil (desde la versión 1.8.6libpcurses.a ya no existe ?).
- **libcurses.a** es la *curses BSD* original, presente en paquetes BSD de distribuciones como la Slackware 2.1.

Las estructuras de datos para la pantalla se llaman ventanas (*windows*) como se define en *ncurses.h*. Una ventana es como un string de caracteres en memoria que el programador puede manipular sin salida al terminal. La ventana por defecto tiene el tamaño del terminal. Puede crear otras ventanas con *newwin(...)*.

Para actualizar el terminal físico de forma óptima, ncurses tiene otra ventana declarada, *curscr*. Esto es una imagen de a qué se parece actualmente el terminal, y *stdscr* es una imagen de como debería parecer el terminal. La salida se efectuará cuando llame a *refresh()*. Ncurses entonces actualizará *curscr* y el terminal físico con la información disponible en *stdscr*. Las funciones de biblioteca usarán optimizaciones internas para actualizar el proceso de forma que pueda cambiar diferentes ventanas y entonces actualizar la pantalla de una vez de una forma óptima.

Con las funciones ncurses puede manipular las estructuras de datos de las ventanas. La funciones que comienzan por *w* le permiten especificar una ventana, mientras que otras afectarán a la ventana. Las funciones que comienzan con *mv* moverán el cursor a la posición *y,x* primero.

Un carácter tiene el tipo *chtype* que es de tipo entero largo sin signo, para almacenar información adicional sobre él (atributos, etc.).

Ncurses usa la base de datos *terminfo*. Normalmente la base de dtos está situada en */usr/lib/terminfo/* y ncurses buscará allí para las definiciones del terminal local. Si quiere comprobar alguna otra definición para una terminal sin cambiar el terminfo original, ponga el valor en la variable de entorno *TERMINFO*. Ncurses comprobará esta variable y usará las definiciones almacenadas allí en lugar de /usr/lib/terminfo/.

La versión actual de ncurses es la 1.8.6().

Al final del capítulo encontrará una tabla con una revisión de las Curses de BSD, NCurses y las Curses de SunOS 5.4. Refiérase a ella cuando quiera buscar una función específica y dónde se encuentra implementada.

8.4 Inicialización

- `WINDOW *initscr()`
  Esta es la primerra función que se normalmente se llama de un programa que usa ncurses. En algunos casos es útil para llamar a **slk_init(int)**, **filter()**, **ripoffline(...)** o **use_env(bf)** antes de **initscr()**. Cuando use terminales múltiples (o quizás capacidades de comprobación), puede usar **newterm(...)** en lugar de **initscr()**.

  initscr() leerá el archivo terminfo apropiado e inicializará la estructura de datos ncurses , reservará memoria para **curscr** y pondrá los valores *LINES* y *COLS* que tiene el terminal. Devolverá un puntero a *stdscr* o ERR cuando ocurra un error. No necesita inicializar el puntero con:

  ```
  stdscr=initscr();
  ```

  initscr() hará esto por usted. Si el valor retornado es ERR, su programa debe salir debido a que ninguna función ncurses funcionará:

  ```
  if(!(initscr())){
    fprintf(stderr,"tipo: initscr() ha fallado\n\n");
    exit (1);
  }
  ```

- `SCREEN *newterm(char *tipo, FILE *outfd, FILE *infd)`
  Para salida por múltiples terminales debe llamarse a **newterm(...)** por cada uno de aquellos que pretenda controlar con ncurses, en lugar de llamar a **initscr()**. El argumento `tipo` es el nombre del terminal, tal como

aparece en la variable de entorno $TERM (ansi, xterm, vt100, etcétera). El argumento `outfd` es el puntero de salida, y el `infd` es el de entrada. Debe llamarse a **endwin()** por cada terminal abierto con **newterm(...)**.

- `SCREEN *set_term(SCREEN *nueva)`
  Com **set_term(SCREEN)** podemos cambiar de terminal. Todas las funciones posteriores actuarán sobre el terminal seleccionado.

- `int endwin()`
  endwin() realiza labores de limpieza, restaura el modo del terminal al estado que tenía antes de llamar a **initscr()** y lleva el cursor a la esquina inferior izquierda. No olvide cerrar todas las ventanas antes de llamar a esta función para finalizar su aplicación.

  Una llamada adicional a **refresh()** después de **endwin()** restaurará el terminal al estado que tuviera antes de llamar a **initscr()** (modo visual); en otro caso será limpiado (modo no visual).

- `int isendwin()`
  Devuelve TRUE si **endwin()** y **refresh()** han sido ejecutadas ya. En otro caso devolverá FALSE.

- `void delscreen(SCREEN* pantalla)`
  Tras llamar a **endwin()**, llámese a **delscreen(SCREEN)** para liberar los recursos, cuando la pantalla del argumento ya no se necesite. (**Nota:** no implementado aun.)

8.5 Ventanas

Las ventanas se pueden crear, borrar, mover, copiar, duplicar y más.

- `WINDOW *newwin(nlineas, ncols, iniy, inix)`
  `iniy` e `inix` son las coordenadas de la esquina superior izquierda de la ventana. `nlineas` es un entero con el número de líneas, y `ncols` es otro entero con el número de columnas.

  ```
  WINDOW *miventana;
  miventana=newwin(10,60,10,10);
  ```

  La esquina superior izquierda de nuestra ventana queda en la línea y columna 10; y tiene 10 líneas y 60 columnas. Si `nlineas` fuera cero, la ventana tendría $LINEAS - iniy$ filas. De la misma manera, tendremos $COLS - inix$ columnas si `ncols` vale cero.

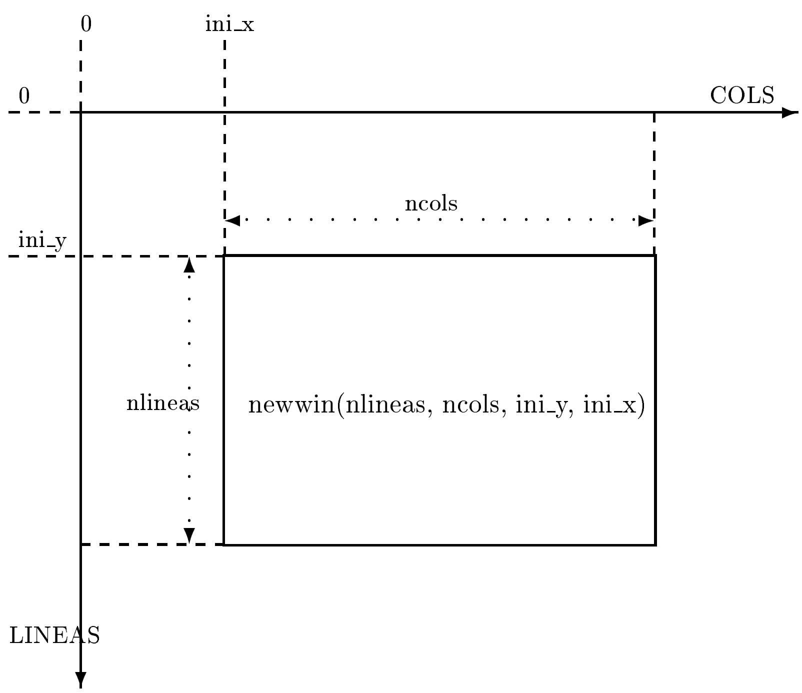
  Cuando llame a **newwin(...)** con todos los argumentos a cero:

  ```
  WINDOW *miventana;
  miventana=newwin(0,0,0,0);
  ```

  la ventana así creada tendrá el tamaño de la pantalla.

  Con la ayuda de *LINES* y *COLS* podemos abrir ventanas en medio de la pantalla, con el código siguiente:

Figura 8.1: Ncurses - esquema de newwin




```
#define MILINEA (int) ((LINES-22)/2)
#define MICOL ((COLS-70)/2)
#define MISLINEAS 22
#define MISCOLS  70
...
WINDOW *vent;
...
if(!(initscr())){
  fprintf(stderr,"tipo: initscr() ha fallado\n\n");
  exit(1);
}
...
if ((LINES<22)||(COLS<70)){
  fprintf(stderr,"pantalla demasiado peque~na\n\n");
  endwin(); exit (1);
}
win=newwin(MISLINEAS,MISCOLS,MILINEA,MICOL);
...
```

Esto abrirá una ventana de 22 líneas y 70 columnas en medio de la pantalla. Comprueba antes que quepa. En la consola de Linux tenemos 25 o más líneas, y 80 o más columnas, pero en los *xterms* este no es el caso, pues son libremente dimensionables.

Alternativamente podemos usar *LINES* y *COLS* para adaptar las ventanas al tamaño de la pantalla:

```
#define MISFILAS   (int) (LINES/2+LINES/4)
#define MISCOLS   (int) (COLS/2+COLS/4)
#define FILAIZ  (int) ((LINES-MISFILAS)/2)
#define COLIZ  (int) (((COLS-2)-MISCOLS)/2)
#define FILADR (int) (FILAIZ)
#define COLDR (int) (FILAIZ+2+MISCOLS)
#define VCOLS    (int) (MISCOLS/2)
...
WINDOW *ventizq, *ventder;
...
ventizq=newwin(MISFILAS, VCOLS, FILAIZ, COLIZ);
ventder=newwin(MISFILAS, VCOLS, FILADR, COLDR);
...
```

Véase *screen.c* en el directorio de ejemplos para más detalle.

- `int delwin(ventana)`
  Borra la ventana `ventana`. Cuando hay subventanas dentro, las borra antes. Además libera todos los recursos que ocupe la ventana. Y Borra todas las ventanas abiertas antes de llamar a **endwin()**.

- `int mvwin(ventana, by, bx)`
  Esta función mueve la ventana a las coordenadas (`by,bx`). Si esto implica mover la ventana más allá de los extremos de la pantalla, no se hace nada y se devuelve ERR.

- `WINDOW *subwin(venorig, nlineas, ncols, iniy, inix)`
  Devuelve una subventana interior a `venorig`. Cuando cambie una de las dos ventanas (la subventana o la otra), este cambio será reflejado en ambas. Llame a **touchwin(venorig)** antes del siguiente **refresh()**.

  `inix` e `iniy` son relativos a la pantalla, no a la ventana `venorig`.

- `WINDOW *derwin(venorig, nlineas, ncols, iniy, inix)`
  Es parecida a la anterior función, solo que ahora los parámetros `iniy` e `inix` son relativos a la ventana `venorig` y no a la pantalla.

- `int mvderwin(ventana, y, x)`
  Mueve la `ventana` deltro de la ventana madre. (**Nota:** no implementado aun.)

- `WINDOW *dupwin(ventana)`
  Duplica la `ventana`.

- `int syncok(ventana, bf)`
  `void wsyncup(ventana)`
  `void wcursyncup(ventana)`
  `void wsyncdown(ventana)`
  (**Nota:** no implementado aun.)

- `int overlay(vent1, vent2)`
  `int overwrite(vent1, vent2)`

overlay(...) copia todo el texto de `vent1` a `vent2` sin copiar los blancos. La función **overwrite(...)** hace lo mismo pero además copia los blancos.

- `int copywin(vent1, vent2, sminfil, smincol, dminfil,`
  `dmincol,`
  `dmaxfil, dmaxcol, overlay)`
  Es similar a **overlay(...)** y **overwrite(...)**, pero proporciona control sobre la región de la ventana a copiar.

8.6 Salida

- `int addch(ch)`
  `int waddch(ven, ch)`
  `int mvaddch(y, x, ch)`
  `int mvwaddch(ven, y, x, ch)`
  Estas funciones se usan para enviar caracteres a la ventana. Para verlos efectivamente habrá que llamar a **refresh()**. Con las funciones **addch()** y **waddch()** se envía el caracter a la ventana actual o a la especificada, respectivamente. Las funciones **mvaddch()** y **mvwaddch()** hacen lo mismo pero previamente mueven el cursor a la posición indicada.

- `int addstr(str)`
  `int addnstr(str, n)`
  `int waddstr(ven, str)`
  `int waddnstr(ven, str, n)`
  `int mvaddstr(y, x, str)`
  `int mvaddnstr(y, x, str, n)`
  `int mvwaddstr(ven, y, x, str)`
  `int mvwaddnstr(ven, y, x, str, n)`

  Estas funciones escriben un string en la ventana y son equivalentes a series de llamadas a **addch()**, etc. `str` debe ser terminado en el carácter nulo (\0). Las funciones con parámetro `ven` especifican la ventana donde escribir. Si no aparece se envía a la ventana estándar (stdscr). Las funciones con parámetro `n` indican cuántos caracteres escribir; y si `n` vale -1, se escribirán todos los caracteres del string.

- `int addchstr(chstr)`
  `int addchnstr(chstr, n)`
  `int waddchstr(ven, chstr)`
  `int waddchnstr(ven, chstr, n)`
  `int mvaddchstr(y, x, chstr)`
  `int mvaddchnstr(y, x, chstr, n)`
  `int mvwaddchstr(ven, y, x, chstr)`
  `int mvwaddchnstr(ven, y, x, chstr, n)`
  Estas funciones copian `chstr` a la ventana. La posición inicial es la del cursor. Las funciones con parámetro `n` escriben esos `n` caracteres del

string; y si vale -1 se escribirán todos. El cursor no es movido ni se comprueban caracteres de control. Estas funciones son más rápidas que las **addstr(...)**. El parámetro `chstr` es un puntero a un array de tipo `chtype`.

- `int echochar(ch)`
  `int wechochar(vent, ch)`
  Es lo mismo que llamar a **addch** o **waddch** seguido de una llamada al **refresh()**.

8.6.1 Salida con Formato

- `int printw(fmt, ...)`
  `int wprintw(win, fmt, ...)`
  `int mvprintw(y, x, fmt, ...)`
  `int mvwprintw(win, y, x, fmt, ...)`
  `int vwprintw(win, fmt, va_list)`
  Estas funciones se corresponden con **printf(...)** y su formas asociadas.

  El paquete **printf(...)** se usa para formatear salidas. Puede definir una cadena de salida e incluir variables de diferentes tipos en ella. Vea la sección 8.1.1 en la página 82 para más información.

  Para usar la función **vwprintw(...)** tiene que incluirse en el programa la cabecera *varargs.h*.

8.6.2 Inserción de Caracteres/Líneas

- `int insch(c)`
  `int winsch(win, c)`
  `int mvinsch(y,x,c)`
  `int mvwinsch(win,y,x,c)`
  El carácter `ch` se inserta a la izquierda del cursor y los demás son movidos una posición a la derecha. El carácter del extremo derecho de la línea puede perderse.

- `int insertln()`
  `int winsertln(win)`
  Inserta una línea en blanco sobre la actual (la línea más inferior se perderá).

- `int insdelln(n)`
  `int winsdelln(win, n)`
  Para valores positivos de `n` estas funciones insertarán n líneas sobre el cursor en la ventana seleccionada, con lo que las n líneas inferiores se perderán. Cuando `n` es negativo, se borrarán n líneas bajo el cursor y las inferiores serán movidas hacia arriba.

- `int insstr(str)`
  `int insnstr(str, n)`

`int winsstr(win, str)`
`int winsnstr(win, str, n)`
`int mvinsstr(y, x, str)`
`int mvinsnstr(y, x, str, n)`
`int mvwinsstr(win, y, x, str)`
`int mvwinsnstr(win, y, x, str, n)`
Estas funciones insertarán la cadena `str` en la línea actual a la izquierda del cursor. Los caracteres de la derecha de éste son movidas a la derecha y se perderán si superan el final de la línea. La posición del cursor no cambia.

`y` y `x` son las coordenadas a las que el cursor será movido antes de insertar la cadena, y `n` es el número de caracteres a insertar (cuando valga 0, se insertará la cadena completa).

8.6.3 Borrado de Caracteres/Líneas

- `int delch()`
  `int wdelch(win)`
  `int mvdelch(y, x)`
  `int mvwdelch(win, y, x)`
  Estas funciones borran el carácter del cursor y mueven los restantes caracteres que estén a la derecha, una posición a la izquierda.

  `y` y `x` son las coordenadas en las que se pondrá el cursor previamente al borrado.

- `int deleteln()`
  `int wdeleteln(win)`
  Estas funciones borran la línea del cursor y mueven las restantes líneas inferiores una posición más arriba.

8.6.4 Cajas y Líneas

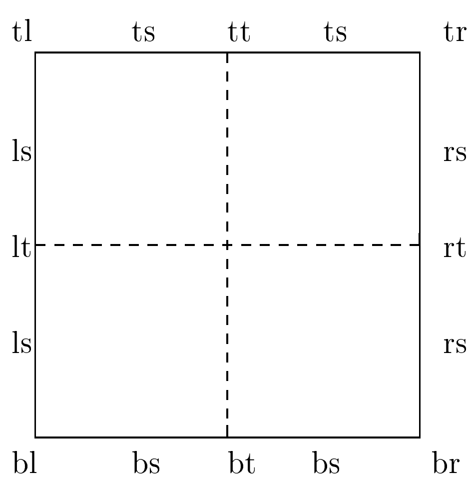
- `int border(ls, rs, ts, bs, tl, tr, bl, br)`
  `int wborder(win, ls, rs, ts, bs, tl, tr, bl, br)`
  `int box(win, vert, hor)`
  Sirven para dibujar un borde en los lados de una ventana (bien sea la *stdscr* o el parámetro `win`). En la siguiente tabla se aprecian los caracteres y sus valores por defecto cuando se llama a **box(...)**. En la figura puede verse la posición de los caracteres en una caja.

- `int vline(ch, n)`
  `int wvline(win, ch, n)`
  `int hline(ch, n)`
  `int whline(win, ch, n)`
  Estas funciones dibujan una línea vertical u horizontal a partir de la posición del cursor. El carácter `ch` esl el que se utiliza, y `n` es el número de caracteres deseados. La posición del cursor no cambia.

Tabla 8.3: Ncurses - caracteres del borde

Carácter	**Posición**	**Defecto**
tl	superior izq.	ACS_ULCORNER
ts	lado superior	ACS_HLINE
tr	superior der.	ACS_URCORNER
ls	lado izquierdo	ACS_VLINE
rs	lado derecho	ACS_VLINE
bl	inferior izq.	ACS_LLCORNER
bs	lado inferior	ACS_HLINE
br	inferior der.	ACS_LRCORNER
rt	centro der.	ACS_RTEE
lt	centro izq.	ACS_LTEE
tt	centro sup.	ACS_TTEE
bt	centro inf.	ACS_BTEE

Figura 8.2: Ncurses - caracteres de caja



8.6.5 Carácter de Fondo

- `void bkgdset(ch)`
  `void wbkgdset(win, ch)`
  Fija el carácter y atributo para la pantalla o una ventana. El atributo en `ch` será ORed con cada carácter no blanco en la ventana. El fondo es entonces parte de la venana y no cambia con desplazamientos ni con las salidas.

- `int bkgd(ch)`
  `int wbkgd(win, ch)`
  Cambia el carácter de fondo y el atributo a `ch`.

8.7 Entrada

- `int getch()`
  `int wgetch(win)`
  `int mvgetch(y, x)`
  `int mvwgetch(win, y, x)`
  getch() leerá la entrada del terminal de una forma que dependerá si el modo de retardo (delay) está activo no. Si delay está activo, **getch()** esperará hasta que se pulse una tecla, en otro caso devolverá la tecla del buffer de entrada o ERR si el buffer está vacío. **mvgetch(...)** y **mvwgetch(...)** moverá primero el cursor a la posición y,x. Las funciones `w` leen la entrada del terminal a la ventana `win`, **getch()** y **mvgetch(...)** del terminal asociado.

  Con **keypad(...)** activado, **getch()** devolverá un código definido en .h como ,macros KEY_* cuando se pulsa una tecla de función. Cuando se pulsa ESCAPE (que puede ser el inicio de una tecla de función) ncurses iniciará un temporizador de un segundo. Si el resto de las pulsaciones no se completa en este segundo, se devuelve la tecla. En otro caso se devuelve el valor de la tecla de función. (Si es necesario, use **notimeout()** para desactivar el temporizador de un segundo).

- `int ungetch(ch)`
  Will put the character `ch` back to the input buffer.

- `int getstr(str)`
  `int wgetstr(win, str)`
  `int mvgetstr(y, x, str)`
  `int mvwgetstr(win, y, x, str)`
  `int wgetnstr(win, str, n)`
  Estas funciones realizarán series de llamadas a **getch()** hasta que se reciba un carácter de fin de línea. Los caracteres se guardan en `str` (por lo que no olvide reservar memoria antes de llamar a estas funciones). Si el eco está activo, la cadena es reflejada en pantalla, y las teclas *kill* y *delete* serán interpretadas (utilice la función **noecho** para desactivar el eco).

- `chtype inch()`
  `chtype winch(win)`
  `chtype mvinch(y, x)`
  `chtype mvwinch(win, y, x)`
  Estas funciones devuelven un carácter de una pantalla o ventana. Como el tipo del valor devuelto es chtype se incluye información de atributo. Esta información se puede extraer del carácter usando las constantes A_* constants (ver tabla 8.4 en la página 112).

- `int instr(str)`
  `int innstr(str, n)`
  `int winstr(win, str)`
  `int winnstr(win, str, n)`
  `int mvinstr(y, x, str)`
  `int mvinnstr(y, x, str, n)`
  `int mvwinstr(win, y, x, str)`
  `int mvwinnstr(win, y, x, str, n)`
  Return a character string from the screen or a window. (**Nota:** no implementado aun.)

- `int inchstr(chstr)`
  `int inchnstr(chstr, n)`
  `int winchstr(win, chstr)`
  `int winchnstr(win, chstr, n)`
  `int mvinchstr(y, x, chstr)`
  `int mvinchnstr(y, x, chstr, n)`
  `int mvwinchstr(win, y, x, chstr)`
  `int mvwinchnstr(win, y, x, chstr, n)`
  Estas funciones devuelven una cadena de carateres de la pantalla o ventana. En la cadena se incluye una información de atributo por cada carácter. (**Nota:** no implementado aun, lib_inchstr no incluida en la librería ncurses.)

8.7.1 Entrada con Formato

- `int scanw(fmt, ...)`
  `int wscanw(win, fmt, ...)`
  `int mvscanw(y, x, fmt, ...)`
  `int mvwscanw(win, y, x, fmt, ...)`
  `int vwscanw(win, fmt, va_list)`
  Estas son similares a **scanf(...)** (vea la sección 8.1.2 en la página 84). **wgetstr(...)** se llama y el resultado se usa como una entrada para scan.

8.8 Opciones

Opciones de Salida

- `int idlok(win, bf)`
  `void idcok(win, bf)`
  Activan o desactivan las características de inserción/borrado del terminal a la ventana; para líneas con **idlok(...)** y para caracteres con **idcok(...)**. (**Nota: idcok(...)** no implementado aun.)

- `void immedok(win, bf)`
  Si es TRUE, cada cambio en la ventana `win` supondrá un refresco de la pantalla física. Esto puede decrementar el rendimiento de un programa, por lo que el valor por defecto es FALSE. (**Nota:** no implementado aun.)

- `int clearok(win, bf)`
  Si `bf` es TRUE, la siguiente llamada a **wrefresh(win)** limpiará la pantalla y la redibujará totalmente (como cuando pulsamos CTRL-L en el editor *vi*).

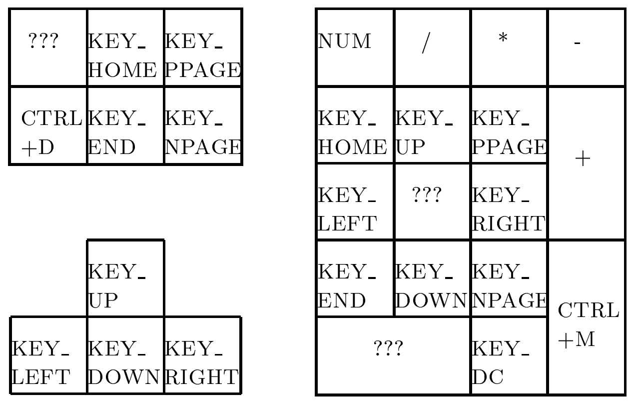
- `int leaveok(win, bf)`
  El comportamiento normal de ncurses deja el cursor físico en el mismo lugar antes del último refresco de la pantalla. Los programas que no usan el cursor pueden ejecutar esta función con el valor TRUE, y evitar el tiempo que requiere mover el cursor. Además, ncurses intentará hacer que el cursor no sea visible.

- `int nl()`
  `int nonl()`
  Controla la traducción del fin de línea. Cuando se activa con la función **nl()**, traducirá el fin de línea a un retorno de carro seguido de una alimentación de línea. Si lo ponemos a OFF con la función **nonl()**, se evitará esta traducción lo que también implica un movimiento del cursor más rápido.

8.8.1 Opciones en la entrada

- `int keypad(win, bf)`
  Si vale TRUE, habilita el teclado numérico de la terminal del usuario cuando está esperando entrada de datos. Ncurses retornará el código de tecla que se define en *ncurses.h* como KEY_* para cada tecla de función y para las teclas con las flechas. Esto es muy útil para un teclado de PC porque se puede de esta manera disponer entonces del bloqueo numérico y de las flechas.

- `int meta(win, bf)`
  Si está en TRUE, los códigos de teclas que retorna **getch()** serán de 8 bits (esto es, no se le pondrá a cero su bit más alto).



- `int cbreak()`
  `int nocbreak()`
  `int crmode()`
  `int nocrmode()`
  cbreak() y **nocbreak()** enciende y apaga, respectivamente el modo CBREAK de la terminal. Cuando CBREAK está encendido, cualquier carácter leído a la entrada estará disponible inmediatamente para el programa; mientras que si está apagado se almacenará en un búfer hasta que aparezca un carácter cambio de línea (*newline*). (**Nota: crmode()** y **nocrmode()** existen sólo por razones de compatibilidad con versiones anteriores, por favor no los utilice..)

- `int raw()`
  `int noraw()`
  Enciende y apaga, respectivamente, el modo RAW. Este modo es igual al CBREAK, excepto por el hecho que en modo RAW no se procesa a los carácteres especiales.

- `int echo()`
  `int noecho()`
  Use **echo()** para obtener eco en pantalla de los carácteres tecleados por el usuario a la entrada, y **noecho()** para que no se vea dicho eco.

- `int halfdelay(t)`
  Como el caso de **cbreak()** pero con una espera de `t` segundos.

- `int nodelay(win, bf)`
  Con TRUE como argumento, configura la terminal en modo inmediato (*non-blocking*). **getch()** retornará ERR si no hay carácteres ya disponibles a la entrada. Si se le da FALSE como argumento, **getch()** esperará hasta que se oprima una tecla.

- `int timeout(t)`
  `int wtimeout(win, t)`

Se recomienda utilizar estas funciones en lugar de **halfdelay(t)** y **nodelay(win,bf)**. El resultado de **getch()** depende del valor de `t`. Si `t` es positivo, la lectura se detiene durante `t` milisegundos, si `t` es cero, no se realiza ninguna detención, y cuando `t` es negativo el programa se detiene hasta que haya carácteres disponibles a la entrada.

- `int notimeout(win, bf)`
  Si `bf` es TRUE, **getch()** utilizará un contador regresivo especial (con un lapso de un segundo) para interpretar y aceptar las secuencias que comienzan con teclas como ESCAPE, etc.

- `int typeahead(fd)`
  Si `fd` es -1 no se realizará control para saber si hay carácteres en espera (*typeahead*); sino, cuando ncurses realice dicho control utilizará el descriptor de fichero `fd` en lugar de *stdin*.

- `int intrflush(win, bf)`
  Cuando se habilita con `bf` en TRUE, entonces cualquiera de las teclas de interrupción que se oprima (quit, break, ...) ocasionará que se exhiban todos los carácteres pendientes de salida en la cola del manejador de la tty.

- `void noqiflush()`
  `void qiflush()`
  (**Nota:** no implementado aun.)

8.8.2 Atributos de la terminal

- `int baudrate()`
  Retorna la velocidad de la terminal en bps (bits per second).

- `char erasechar()`
  Retorna el actual carácter que sirve para borrar (*erase*).

- `char killchar()`
  Retorna el carácter actual para «matar» la línea actual (*kill*).

- `int has_ic()`
  `int has_il()`
  has_ic() retorna TRUE si la terminal tiene la capacidad de insertar/borrar de a un carácter **has_il()** retorna TRUE cuando la terminal tiene la capacidad de insertar/borrar de a líneas. Si no fuera así, las funciones retornan ERR. (**Nota:** no implementado aun.)

- `char *longname()`
  Retorna un apuntador que nos permite acceder a la descripción de la terminal actual.

- `chtype termattrs()`
  (**Nota:** no implementado aun.)

- `char *termname()`
  Retorna el contenido de la variable del entorno de usuario TERM. (**Nota:** no implementado aun.)

8.8.3 ¿Cómo se usa?

Hasta ahora hemos visto las opciones de las ventanas y los modos de las terminales, ya es hora de describir cómo se utiliza todo esto.

En Linux lo primero es habilitar el teclado numérico. Esto permitirá la utilización de las teclas de las flechas y el teclado numérico.

```
keypad(stdscr,TRUE);
```

Ahora bien, existen dos maneras fundamentales de esperar entradas desde el teclado.

1. El progrma quiere que el usuario oprima una tecla y luego en función de la tecla seleccionada se elegirá el procedimiento apropiado. (Por ejemplo, `"Oprima 't' para terminar"` y luego el programa aguarda una *t*)

2. El programa quiere que el usuario escriba una cadena de carácteres dentro de una máscara exhibida en la pantalla. Por ejemplo, un nombre de directorio, o una dirección postal en una base de datos.

Para el primer caso, utilizaremos las siguientes opciones y modos:

```
char c;

noecho();
timeout(-1);
nonl();
cbreak();
keypad(stdscr,TRUE);
while(c=getch()){
  switch(c){
    case 't': funcion_de_terminaci\'on();
    default:  break;
  }
}
```

El programa se detiene hasta que se oprime una tecla. Si la tecla fue *s* llamamos a nuestra función de terminación, sino, esperamos por otra tecla.

La construcción `switch` puede expandirse hasta llenar nuestras necesidades de procesamiento de entradas. Utilice las macros `KEY_*` para leer las teclas especiales, por ejemplo:

```
KEY_UP      KEY_RIGHT    KEY_A1    KEY_B2    KEY_C1
KEY_DOWN    KEY_LEFT     KEY_A3              KEY_C3
```

le servirán para leer las teclas de movimiento de cursor.

Si desea ver el contenido de un fichero, deberá utilizar un código como el que sigue:

```
 int sigo=TRUE;
 char c;
 enum{ARRIBA,ABAJO,DERECHA,IZQUIERDA};

 noecho();
 timeout(-1);
 nonl();
 cbreak();
 keypad(stdscr,TRUE);
 while(sigo==TRUE){
   c=getch();
   switch(c){
     case KEY_UP:
     case 'a':
     case 'A': scroll_s(ARRIBA);
               break;
     case KEY_DOWN:
     case 'b':
     case 'B': scroll_s(ABAJO);
               break;
     case KEY_LEFT:
     case 'i':
     case 'I': scroll_s(IZQUIERDA);
               break;
     case KEY_RIGHT
     case 'd':
     case 'D': scroll_s(DERECHA);
               break;
     case 't':
     case 'T': sigo=FALSE;
     default:  break;
   }
}
```

Para el segundo caso, sólo necesitamos ejecutar **echo()** y entonces los carácteres tecleados por el usuario se escribirán en la pantalla. Para poner los carácteres en alguna posición deseada en particular, utilice **move(...)** o **wmove(...)**.

O sino, podemos abrir una ventana con una máscara en ella (por ejemplo podemos elegir colores distintos para resaltar la máscara) y solicitar al usuario que ingrese una cadena:

```
 WINDOW *maskwin;
 WINDOW *mainwin;
 char *ptr=(char *)malloc(255);
 ...
   mainwin=newwin(3,37,9,21);
   maskwin=newwin(1,21,10,35);
   ...
   werase(mainwin);
   werase(maskwin);
   ...
```

```
box(mainwin,0,0);
mvwaddstr(mainwin,1,2,"Cadena a ingresar: ");
...
wnoutrefresh(mainwin);
wnoutrefresh(maskwin);
doupdate();
...
mvwgetstr(maskwin,0,0,ptr);
...
delwin(maskwin);
delwin(mainwin);
endwin();
free(ptr);
```

Mire por favor *input.c* en el directorio de ejemplos, para una mejor explicación.

8.9 ¿Cómo borrar ventanas y líneas?

- `int erase()`
  `int werase(win)`
  werase(...) y **erase()** taparán con espacios en blanco cada posición de la ventana `win` o de *stdscr*, respectivamente. Por ejemplo, si Ud. configura los atributos de una ventana con ciertos colores y luego llama a **werase()**, la ventana se coloreará. He tenido algunos problemas con `COLOR_PAIRS` cuando defino atributos distintos a negro sobre blanco, así que terminé escrbiendo mi propia función para borrar (que accede a bajo nivel a la estructura `WINDOW`:

  ```
  void NewClear(WINDOW *win)
  {
  int  y,x;

   for ( y = 0 ; y <= win -> _maxy ; y++ )
     for ( x = 0 ; x <= win -> _maxx ; x++ )
       (chtype *) win-> _line[y][x] = ' '|win-> _attrs;
   win -> _curx = win -> _cury = 0;
   touchwin(win);
  }
  ```

  El problema es que ncurses a veces no utiliza los atributos de la ventana cuando limpia la pantalla. Por ejemplo, en *lib_clrtoeol.c*, se define a BLANK como:

  ```
  #define BLANK ' '|A_NORMAL
  ```

  así que los otros atributos se pierden al borrar hasta el final de la línea.

- `int clear()`
  `int wclear(win)`
  Igual que **erase()**, pero ejecuta además **clearok()** (la pantalla se limpiará al realizarse el siguiente refresco).

- `int clrtobot()`
  `int wclrtobot(win)`
  Borra la línea donde se encuentra el cursor, comenzando desde el carácter justo a la derecha del cursor, y la línea debajo del cursor.

- `int clrtoeol()`
  `int wclrtoeol(win)`
  Borra la línea actual desde la posición del cursor hasta el final.

8.10 Actualización de la imagen an la terminal

Como ya se mencionó en la introducción, las ventanas de ncurses son imágenes en memoria. Esto significa que cualquier cambio que se realice en una ventana no se refleja en la pantalla física de la terminal hasta que se efectúe un «refresco». De esta manera se optimiza la tarea de enviar la salida a la terminal porque se puede realizar un montón de cambios y luego, de una sola vez, llamar a `refresh()` para que escriba la pantalla final. Si no se manejara de este modo, cada pequeño cambio en la pantalla debería enviarse a la terminal, y por lo tanto perjudicaría la performance del programa del usuario.

- `int refresh()`
  `int wrefresh(win)`
  refresh() copia *stdscr* a la terminal y **wrefresh(win)** copia la imagen de la ventana a *stdscr* y luego hace que *curscr* se vea como *stdscr*.

- `int wnoutrefresh(win)`
  `int doupdate()`
  wnoutrefresh(win) copia sólo la ventana `win` a *stdscr*. Esto significa no se ha realizado ninguna actualización de la terminal, aunque la pantalla virtual *stdscr* tiene la disposición actualizada. **doupdate()** se ocupará de enviar la salida a la terminal. De esta manera, un programa puede cambiar varias ventanas, llamar a **wnoutrefresh(win)** para cada ventana y luego llamar a **doupdate()** para actualizar la pantalla física sólo una vez.

  Por ejemplo, tenemos el siguiente programa con dos ventanas. Procedemos a alterar ambas ventanas cambiando algunas líneas de texto. Podemos escribir *changewin(win)* con **wrefresh(win)**.

```
main()                         changewin(WINDOW *win)
{                              {
WINDOW *win1,*win2;                ... /* aqu\'{\i} alteramos */
    ...                            ... /* las l\'{\i}neas */
    changewin(win1);               wrefresh(win);
    changewin(win2);           return;
```

```
    ...                    }
}
```

De esta manera, ncurses deberá actualizar dos veces la terminal, y por lo tanto disminuirá la velocidad de ejecución de nuestro programa. Con **doupdate()** modificamos *changewin(win)* y la funcion `main()` obteniendo una mejor performance.

```
main()                    changewin(WINDOW *win)
{                         {
WINDOW *win1,*win2;           ... /* aqu\'{\i} alteramos */
    ...                       ... /* las l\'{\i}neas */
    changewin(win1);          wnoutrefresh(win);
    changewin(win2);      return;
    doupdate();           }
    ...
}
```

- `int redrawwin(win)`
  `int wredrawln(win, bline, nlines)`
  Utilice estas funciones cuando algunas líneas o toda la pantalla deba descartarse antes de escribir algo nuevo (puede ser por ejemplo cuando las líneas en la pantalla se han mezclado con basura, o algo así).

- `int touchwin(win)`
  `int touchline(win, start, count)`
  `int wtouchln(win, y, n, changed)`
  `int untouchwin(win)`
  Le indica a ncurses que toda la ventana `win` o las líneas que van desde la `start` hasta la `start+count` se han tocado. Por ejemplo, cuando tiene algunas ventanas que se solapan (como en el ejemplo de *type.c*) y se produce un cambio en una ventana, no se afecta a la imagen de la otra.

  wtouchln(...) marcará como tocadas las `n` líneas que comienzan en `y`. Si `change` se pone en TRUE, entonces se marcan como tocadas dichas líneas, sino se marcan como que no han sido tocadas (cambiadas o no cambiadas).

  untouchwin(win) marcará la ventana `win` como que no ha sido modificada desde la última llamada a **refresh()**.

- `int is_linetouched(win, line)`
  `int is_wintouched(win)`
  Con estas funciones, Ud. puede controlar si la línea `line` o la ventana `win` ha sido tocada desde la última llamada a **refresh()**.

8.11 Atributos de vídeo y colores

Los atributos son capacidades especiales de la terminal que se utilizan al escribir los carácteres en la pantalla. Los carácteres pueden escribirse en negrilla (*bold*),

Tabla 8.4: Ncurses - atributos

Definición	**Atributo**
A_ATTRIBUTES	máscara para los atributos (chtype)
A_NORMAL	normal, quita todos los otros
A_STANDOUT	el mejor modo para resaltar
A_UNDERLINE	subrayado
A_REVERSE	vídeo en reverso
A_BLINK	parpadeante
A_DIM	brillo disminuído o medio brillo
A_BOLD	negrilla o brillo extra
A_ALTCHARSET	usar conjunto de carácteres alternativos
A_INVIS	invisible
A_PROTECT	???
A_CHARTEXT	máscara para el carácter actual (chtype)
A_COLOR	máscara para el color
COLOR_PAIR(n)	que el par de colores sea el almacenado en n
PAIR_NUMBER(a)	obtener el par de colores almacenado en el atributo a

subrayado, parpadeantes, etc.. Con ncurses, Ud. disfruta de la posibilidad de encender y apagar estos atributos y de esa manera puede mejorar la apariencia de la salida. Los posibles atributos se enumeran en la siguiente tabla:

Ncurses define ocho colores que Ud. puede utilizar en una terminal que puede mostrar colores. Primero, inicialice las estructuras de datos para color con **start_color()**, luego controle la existencia de las capacidades de color con **has_colors()**. **start_color()** inicializará *COLORS*, la máxima cantidad de colores que puede manejar la terminal, y *COLOR_PAIR*, la máxima cantidad de pares de colores que podrá definir.

Los atributos se pueden combinar con el operador OR «'|», así que puede obtener negrilla y parpadeante mediante:

```
A_BOLD|A_BLINK
```

Tabla 8.5: Ncurses - colores

Definición	**Color**
COLOR_BLACK	negro
COLOR_RED	rojo
COLOR_GREEN	verde
COLOR_YELLOW	amarillo
COLOR_BLUE	azul
COLOR_MAGENTA	magenta
COLOR_CYAN	cyan
COLOR_WHITE	blanco

Cuando se asignan ciertos atributos `attr` a una ventana, todos los carácteres que escriba en dicha vantana se mostrarán con esas propiedades, hasta haga un cambio en los atributos de la ventana. No se perderán cuando enrolle (`scroll`) la ventana, ni cuando la mueva, o accione sobre ella de cualquier otra manera.

Si Ud. escribe programas que pueden utilizar ncurses y BSD curses, recuerde que la BSD curses no permite el uso de colores. (Tampoco hay colores en las versiones antiguas de ncurses tipo SYS V.) Así que si desea utilizar ambas bibliotecas, deberá utilizar estructuras de compilación condicional con *#ifdef*.

- `int attroff(attr)`
  `int wattroff(win, attr)`
  `int attron(attr)`
  `int wattron(win, attr)`
  Encienden (on) o apagan (off) el atributo especificado mediante `attr` sin tocar los otros atributos en la ventana (que será *stdscr* o `win`).

- `int attrset(attr)`
  `int wattrset(win, attr)`
  Hace que los atributos de *stdscr* o `win` se configuren en `attr`.

- `int standout()`
  `int standend()`
  `int wstandout(win)`
  `int wstandend(win)`
  Enciende y apaga el modo `standout` sobre la ventana (*stdscr* o `win`), que se utiliza para resaltar texto.

- `chtype getattrs(win)`
  Retorna los atributos que tiene `win` al momento de la llamada a esta función.

- `bool has_colors()`
  Retorna TRUE si la terminal tiene colores. Antes de utilizar colores, Ud. debe controlar con **has_colors()** que la terminal los pueda manejar, y a continuación debe inicializar los colores con **start_color())**.

- `bool can_change_color()`
  TRUE si la terminal puede redefinir colores.

- `int start_color()`
  Inicialización de colores. Debe llamar a esta función antes de utilizar el manejo de colores!

- `int init_pair(pair, fg, bg)`
  Cuando en los argumentos a funciones de ncurses, donde se espera un atributo queremos poner colores, debemos utilizar los **pares de colores**. La definición de un par de colores se realiza con **init_pair(...)**. `fg` es el color del primer plano (carácteres) y `bg` es el color del fondo que se asocian en el par de colores `pair`. El par de colores `pair` no es más que

un número en el rango de 1 a $COLORPAIRS-1$ (Si, leyó bien, desde el 1; pues el 0 está reservado para el par negro sobre blanco. Una vez que ha sido definido, el `pair` se comporta como un atributo. Por ejemplo, si desea poner carácteres rojos sobre fondo azul, haga:

```
init_pair(1,COLOR_RED,COLOR_BLUE);
```

Ahora puede llamar a **wattr(...)** para que `win` tenga como colores los de nustro nuevo par:

```
wattr(win,COLOR_PAIR(1));
```

O puede combinar pares de colores con otros atributos, como se muestra a continuación:

```
wattr(win ,A_BOLD|COLOR_PAIR(1));
wattr(win1,A_STANDOUT|COLOR_PAIR(1));
```

El primero pone los colores que habíamos seleccionado y además enciende el atributo `BOLD`; el segundo ejemplo pone los colores y además levanta el brillo (`STANDOUT`), así que obtenemos rojo brillante sobre azul.

- `int pair_content(pair, f, b)`
  Obtiene los colores de primer plano (`f`) y fondo (`b`) correspondientes al par de colores `pair`.

- `int init_color(color, r, g, b)`
  Cambia los componentes del color `color`. Los componentes son `r` (rojo), `g` (verde) and `b` (azul), y pueden tomar valores en el rango 1 a $COLORS-1$.

- `int color_content(color, r, g, b)`
  Devuelve los componentes `r` (rojo), `g` (verde) y `b` (azul) que forman un dado `color`.

Bueno, la pregunta ahora será: cómo combinar atributos y colores?. Algunas terminales, como la consola de Linux, tienen colores; otras, como xterm, vt100, no los tienen. El código que se muestra a continuación debería resolver este problema:

```
void CheckColor(WINDOW *win1, WINDOW *win2)
{
 start_color();
 if (has_colors()){
   /* muy bien, tenemos colores, as\'{\i} que definimos los pares de
    * colores para car\'acter y para fondo.
    */
   init_pair(1,COLOR_BLUE,COLOR_WHITE);
   init_pair(2,COLOR_WHITE,COLOR_RED);
   /* ahora usamos los pares de colores reci\'en definidos para
    * configurar las ventanas
    */
```

```
    wattrset(win1,COLOR_PAIR(2));
    wattrset(win2,COLOR_PAIR(1));
  }
  else{
    /* Arf!, no hay colores (a lo mejor estamos en una vt100 o xterm).
     * Bien, entonces utilizaremos negro sobre blanco
     */
    wattrset(win1,A_REVERSE);
    wattrset(win2,A_BOLD);
  }
return;
}
```

Primero, la función *CheckColor* inicializa los colores con **start_color()**, luego la función **has_colors()** retornará TRUE si la terminal puede mostrar colores. Si nos encontramos que acepta colores, llamamos a **init_pair(...)** para combinar los colores de frente con fondo en un par de colores, y luego llamamos a

wattrset(...) para configurar las ventanas con los colores correspondientes. En el caso en que no tuviéramos la posibilidad de colores en nuestra terminal, nos alcanza con utilizar **wattrset(...)** para poner los atributos que tolera nuestra terminal monocroma.

Para obtener colores en xterm, la mejor manera que he encontrado consiste en utilizar la ansi_xterm con las entradas emparchadas correspondientes al terminfo del Midnight Commander. Si Ud. quiere usar la misma solución, consiga los fuentes de ansi_xterm y Midnight Commander (mc-x.x.tar.gz); compile la ansi_xterm; use tic con xterm.ti y vt100.ti que obtiene del archivo mc-x.x.tar.gz; ejecute ansi_xterm y compruebe su funcionamiento.

8.12 Coordenadas del cursor y de las ventanas

- `int move(y, x)`
  `int wmove(win, y, x)`
  move() mueve el cursor dentro de *stdscr*, **wmove(win)** mueve el cursor dentro de la ventana `win`. Para las funciones de entrada/salida, se definen macros adicionales que mueven el cursor antes de llamar a la función especificada.

- `int curs_set(bf)`
  Muestra u oculta el cursor, si la terminal es capaz de esta operación.

- `void getyx(win, y, x)`
  getyx(...) devuelve la posición del cursor al momento de la llamada. (**Nota:** Es una macro.)

- `void getparyx(win, y, x)`
  Cuando `win` es una subventana, **getparyx(...)** nos entregará las coordenadas de la ventana en relación a su ventana paterna, almacenándolas

en `y` y `x`. En cualquier otro caso `y` y `x` se pondrán a -1. (**Nota:** no implementado aun.)

- `void getbegyx(win, y, x)`
  `void getmaxyx(win, y, x)`
  `int getmaxx(win)`
  `int getmaxy(win)`
  Guardan en `y` y `x` las coordenadas de posición y tamaño de `win`.

- `int getsyx(int y, int x)`
  `int setsyx(int y, int x)`
  Almacena la posición del cursor dentro de la pantalla virtual en `y` y `x` o lo posiciona allí, respectivamente. Si pone a -1 los valores de `y` y `x` y llama a **getsyx(...)**, se habilitará *leaveok*.

8.13 Moviéndonos por allí

- `int scrollok(win, bf)`
  Si se pone a TRUE, entonces el texto en la ventana `win` se moverá una línea hacia arriba cuando se escriba un carácter (o un cambio de línea) y el cursor estaba posicionado sobre el carácter de la esquina inferior derecha. Si se pone a FALSE, el cursor quedará en la misma posición.

  Cuando se habilita (con TRUE), se puede mover el contenido de las ventanas mediante la utilización de las siguientes funciones. (**Nota:** Las líneas del contenido de la ventana también se moverán si escribe un cambio de línea en la última línea de la ventana. Así que tenga cuidado con **scrollok(...)** o le soprenderán los resultados..)

- `int scroll(win)`
  Mueve las líneas de la ventana (y en la estructura de datos) una línea hacia ariba.

- `int scrl(n)`
  `int wscrl(win, n)`
  Estas funciones mueven la pantalla *stdscr* o la ventana `win` hacia arriba o hacia abajo, de acuerdo al valor del entero `n`. Si `n` es positivo, las líneas de la ventana se mueven `n` líneas hacia arriba, si `n` es negativo se moverá hacia abajo `n` líneas.

- `int setscrreg(t, b)`
  `int wsetscrreg(win, t, b)`
  Configura una región de movimientos por software.

El código que se muestra a continuación le mostrará cómo puede obtener el efecto de movimiento de las líneas de texto en la pantalla. Vea además en *type.c* en el directorio de los ejemplos.

Tenemos una ventana con 18 líneas y 66 columnas, en la cual queremos mover el texto. *s*[] es un vector de carácteres con el texto. *max_s* es el número de la última línea en *s*[]. *clear_line* escribe carácteres blancos desde la posición actual del cursor hasta el fin de la línea, y utiliza los atributos actuales en la ventana (y no con el atributo A_NORMAL como lo hace `clrtoeol()`. *beg* es la última línea de

s[] que se muestra en la pantalla en cualquier momento dado. *scroll* es un tipo enumerado para indicar a la función qué es lo que hay que hacer: si mostrar la línea SIGuiente o la ANTerior.

```
enum{ANT,SIG)};

void scroll_s(WINDOW *win, int scroll)
{
 /* verificar si necesitamos mover las l\'{\i}neas,
  * y si hay l\'{\i}neas para mover
  */
 if((scroll==SIG)&&(beg<=(max_s-18))){
 /* una l\'{\i}nea para abajo */
     beg++;
 /* habilitar el movimiento */
     scrollok(win, TRUE);
 /* mover */
     wscrl(win, +1);
 /* deshabilitar el movimiento */
     scrollok(win, FALSE);
 /* colocar la cadena de car\'acteres de la \'ultima l\'{\i}nea */
     mvwaddnstr(win,17,0,s[beg+17],66);
 /* limpiar la \'ultima l\'{\i}nea despu\'es del \'ultimo car\'acter ocupado
  * y hasta el fin de l\'{\i}nea.
  * Si no se hace, los atributos se ver\'an mal
  */
     clear_line(66,win);
 }
 else if((scroll==ANT)&&(beg>0)){
     beg--;
     scrollok(win, TRUE);
     wscrl(win, -1);
     scrollok(win, FALSE);
     mvwaddnstr(win,0,0,s[beg],66);
     clear_line(66,win);
 }
 wrefresh(win);
return;
}
```

8.14 Pads

- `WINDOW *newpad(nlines, ncols)`

- `WINDOW *subpad(orig, nlines, ncols, begy, begx)`

- `int prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)`

- `int pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)`

- `int pechochar(pad, ch)`

8.15 Soft-labels

- `int slk_init(int fmt)`

- `int slk_set(int labnum, char *label, int fmt)`

- `int slk_refresh()`

- `int slk_noutrefresh()`

- `char *slk_label(int labnum)`

- `int slk_clear()`

- `int slk_restore()`

- `int slk_touch()`

- `int slk_attron(chtype attr)`
  `int slk_attrset(chtype attr)`
  `int slk_attroff(chtype attr)`
  Estas funciones corresponden a **attron(attr)**, **attrset(attr)** y **attroff(attr)**. No se han construído aún.

8.16 Miscelánea

- `int beep()`

- `int flash()`

- `char *unctrl(chtype c)`
- `char *keyname(int c)`
- `int filter()`
  (**Nota:** no implementado aun.)
- `void use_env(bf)`
- `int putwin(WINDOW *win, FILE *filep)`
  (**Nota:** no implementado aun.)
- `WINDOW *getwin(FILE *filep)`
  (**Nota:** no implementado aun.)
- `int delay_output(int ms)`
- `int flushinp()`

8.17 Acceso de Bajo Nivél

- `int def_prog_mode()`
- `int def_shell_mode()`
- `int reset_prog_mode()`
- `int reset_shell_mode()`
- `int resetty()`
- `int savetty()`
- `int ripoffline(int line, int (*init)(WINDOW *, int))`
- `int napms(int ms)`

8.18 Volcado de Pantalla

- `int scr_dump(char *filename)`
  (**Nota:** no implementado aun.)
- `int scr_restore(char *filename)`
  (**Nota:** no implementado aun.)
- `int scr_init(char *filename)`
  (**Nota:** no implementado aun.)
- `int scr_set(char *filename)`
  (**Nota:** no implementado aun.)

8.19 Emulación Termcap

- `int tgetent(char *bp, char *name)`
- `int tgetflag(char id[2])`
- `int tgetnum(char id[2])`
- `char *tgetstr(char id[2], char **area)`
- `char *tgoto(char *cap, int col, int row)`
- `int tputs(char *str, int affcnt, int (*putc)())`

8.20 Funciones Terminfo

- `int setupterm(char *term, int fildes, int *errret)`
- `int setterm(char *term)`
- `int set_curterm(TERMINAL *nterm)`
- `int del_curterm(TERMINAL *oterm)`
- `int restartterm(char *term, int fildes, int *errret)`
  (**Nota:** no implementado aun.)

- `char *tparm(char *str, p1, p2, p3, p4, p5, p6, p7, p8, p9)`
  p1 - p9 long int.
- `int tputs(char *str, int affcnt, int (*putc)(char))`
- `int putp(char *str)`
- `int vidputs(chtype attr, int (*putc)(char))`
- `int vidattr(chtype attr)`
- `int mvcur(int oldrow, int oldcol, int newrow, int newcol)`
- `int tigetflag(char *capname)`
- `int tigetnum(char *capname)`
- `int tigetstr(char *capname)`

8.21 Funciones de Depurado

- `void _init_trace()`
- `void _tracef(char *, ...)`
- `char *_traceattr(mode)`
- `void traceon()`
- `void traceoff()`

8.22 Atributos Terminfo

8.22.1 Atributos Lógicos

Variable	**Nombre Car.**	**Cód. Int.**	**Descripción**
auto_left_margin	`bw`	`bw`	cub1 ajusta de la columna 0 a la última

auto_right_margin	am	am	La terminal tiene márgenes automáticos
back_color_erase	bce	ut	Borrado de pantalla con el color del segundo plano
can_change	ccc	cc	La terminal puede redefinir los colores existentes
ceol_standout_glitch	xhp	xs	Los caracteres destacados no se borran al sobresecibirse (hp)
col_addr_glitch	xhpa	YA	Sólo se permite movimientos positivos para hpa/mhpa
cpi_changes_res	cpix	YF	El cambio de la frecuencia de puntos de un caracter cambia la resolución
cr_cancels_micro_mode	crxm	YB	El uso del retorno de carro (cr) sale del modo micro
eat_newline_glitch	xenl	xn	El caracter de nueva línea (nl) se ignora pasadas las 80 cols (Concepto)
erase_overstrike	eo	eo	Se pueden borrar los sobreimpresionados con un blanco
generic_type	gn	gn	Tipo de línea generérico (p.ej., llamada, conmutador).
hard_copy	hc	hc	La terminal que produce copia sobre papel
hard_cursor	chts	HC	El cursor es difícil de ver
has_meta_key	km	km	Tiene la tecla meta (shift activa el bit de paridad)
has_print_wheel	daisy	YC	La impresora necesita un operador para cambiar de conjunto de caracteres
has_status_line	hs	hs	Tiene ”línea de estado” extra
hue_lightness_saturation	hls	hl	La terminal utiliza únicamente la notación HLS para color (Tektronix)
insert_null_glitch	in	in	El modo de inserción distingue los caracteres nulos
lpi_changes_res	lpix	YG	El cambio del la frecuencia de puntos de la línea cambia la resolución
memory_above	da	da	Puede almacenar información encima de la pantalla
memory_below	db	db	Puede almacenar información debajo de la pantalla
move_insert_mode	mir	mi	No es arriesgado moverse durante la inserción
move_standout_mode	msgr	ms	No es arriesgado moverse en los modo destacados
needs_xon_xoff	nxon	nx	No vale rellenar, se requiere xon/xoff
no_esc_ctl_c	xsb	xb	Colmena (f1=escape, f2=ctrl C)
non_rev_rmcup	nrrmc	NR	smcup no deshace rmcup
no_pad_char	npc	NP	No existe caracter de relleno
non_dest_scroll_region	ndscr	ND	La región de paginado no destruye la información
over_strike	os	os	La terminal sobreescribe
prtr_silent	mc5i	5i	La impresora no envía eco a la pantalla
row_addr_glitch	xvpa	YD	Sólo se permite movimientos positivos para vhp/mvpa
semi_auto_right_margin	sam	YE	La escritura en la última columna resulta en un returno de carro
status_line_esc_ok	eslok	es	Se puede usar el caracter de escape en la línea de estado
dest_tabs_magic_smso	xt	xt	Los tabuladores arrunian, magic so char (Teleray 1061)
tilde_glitch	hz	hz	Hazel-tine; no se puede imprimir el símbolo de tilde
transparent_underline	ul	ul	El caracter de subrayado se sobreescribe
xon_xoff	xon	xo	La terminal usa el protocolo xon/xoff

8.22.2 Números

Variable	**Nombre Car.**	**Cód. Int.**	**Descripción**
bit_image_entwining	bitwin	Yo	No está documentado en el SYSV
buffer_capacity	bufsz	Ya	Número de bytes almacenados antes de imprimir
columns	cols	co	Número de columnas por línea
dot_vert_spacing	spinv	Yb	Espaciado horizonal de los puntos en puntos por pulgada
dot_horz_spacing	spinh	Yc	Espaciado vertical de pines en pines por pulgada
init_tabs	it	it	Tabuladores iniclamente cada # de espacios
label_height	lh	lh	Filas por etiqueta
label_width	lw	lw	Columnas por etiqueta
lines	lines	li	Número de líneas por pantalla o página
lines_of_memory	lm	lm	Número de líneas en la memoria. 0 indica que varía.

magic_cookie_glitch	`xmc`	`sg`	Número de espacios que producen smso o rmso
max_colors	`colors`	`Co`	Máximo número de colores en la pantalla
max_micro_address	`maddr`	`Yd`	Valor máximo de las micro_... direcciones
max_micro_jump	`mjump`	`Ye`	Valor máximo de parm_..._micro
max_pairs	`pairs`	`pa`	Número máximo de parejas de color en la pantalla
micro_col_size	`mcs`	`Yf`	Tamaño del paso de caracter en modo micro
micro_line_size	`mls`	`Yg`	Tamaño del paso de línea en modo micro
no_color_video	`ncv`	`NC`	Atributos de vídeo que no se pueden usar en colores
number_of_pins	`npins`	`Yh`	Número de pines en la cabeza de impresión
num_labels	`nlab`	`Nl`	Número de etiquetas en la pantalla
output_res_char	`orc`	`Yi`	Resolución horizontal en unidades por línea
output_res_line	`orl`	`Yj`	Resolución vertical en unidades por línea
output_res_horz_inch	`orhi`	`Yk`	Resolución horizontal en unidades por pulgada
output_res_vert_inch	`orvi`	`Yl`	Resolución vertical en unidades por pulgada
padding_baud_rate	`pb`	`pb`	Mínimo numero de baudios que necesita relleno de cr/nl
virtual_terminal	`vt`	`vt`	Número de terminal virtual (Sistema UNIX)
width_status_line	`wsl`	`ws`	Número de columnas en la línea de estado

(Los siguientes atributos numericos están presentes en la estructura term del SYSV, pero no se han documentado aun en la página de manual. Los comentarios provienen del fichero de cabecera que contiene la definición de la estructura.)

bit_image_type	`bitype`	`Yp`	Tipo de dispositivo de imágenes por bit
buttons	`btns`	`BT`	Número de botones por ratón
max_attributes	`ma`	`ma`	Número máximo de atributos que la terminal puede manejar
maximum_windows	`wnum`	`MW`	Número máximo de vetanas definibles
print_rate	`cps`	`Ym`	Velocidad de impresión en caracteres por segundo
wide_char_size	`widcs`	`Yn`	Tamaño del paso de un caracter en modo doble ancho

8.22.3 Cadenas

Variable	**Nombre Car.**	**Cód. Int.**	**Descripción**
acs_chars	`acsc`	`ac`	Parejas de conjuntos de caracteres gráficos - por defecto vt100
alt_scancode_esc	`scesa`	`S8`	Escape alternativo para emulación del código escaneado (por defecto setoma vt100)
back_tab	`cbt`	`bt`	Tabulador inverso (P)
bell	`bel`	`bl`	Señal audible (timbre) (P)
bit_image_repeat	`birep`	`Xy`	Repetir la célula de imagen por bits #1, #2 veces (usar tparm)
bit_image_newline	`binel`	`Zz`	Desplazarse hasta la siguiente fila de la imagen por bits (usar tparm)
bit_image_carriage_return	`bicr`	`Yv`	Desplazarse hasta el comienzo de esta fila (usar tparm)
carriage_return	`cr`	`cr`	Retorno de carro (P*)
change_char_pitch	`cpi`	`ZA`	Cambia # de caracteres por pulgada
change_line_pitch	`lpi`	`ZB`	Cambia # de líneas por pulgada
change_res_horz	`chr`	`ZC`	Cambia la resolución horizontal
change_res_vert	`cvr`	`ZD`	Cambia la resolución vertical

change_scroll_region	`csr`	`cs`	Cambia de las líneas #1 a la #2 (vt100) (PG)
char_padding	`rmp`	`rP`	Como ip pero cuando se está en modo inserción
char_set_names	`csnm`	`Zy`	Lista de los nombres de conjuntos de caracteres
clear_all_tabs	`tbc`	`ct`	Borra todos las paradas del tabulador (P)
clear_margins	`mgc`	`MC`	Borra todos los márgenes (superior, inferior y laterales)
clear_screen	`clear`	`cl`	Borra la pantalla y desplaza el cursor al comienzo (P*)
clr_bol	`el1`	`cb`	Borra hasta el comienzo de la línea
clr_eol	`el`	`ce`	Borra hasta el final de la línea (P)
clr_eos	`ed`	`cd`	Borra hasta el final de la pantalla (P*)
code_set_init	`csin`	`ci`	Secuencia de inicio para conjuntos de códigos múltiples
color_names	`colornm`	`Yw`	Da un nombre al color #1
column_address	`hpa`	`ch`	Fija la columna del cursor (PG)
command_character	`cmdch`	`CC`	Caracter de cmd se puede fijar por la terminal en el prototipo
cursor_address	`cup`	`cm`	Desplazamiento relativo del cursor fila #1 columna #2 (PG)
cursor_down	`cud1`	`do`	Baja una línea
cursor_home	`home`	`ho`	Desplaza el cursor al inicio (sin cup)
cursor_invisible	`civis`	`vi`	Hace el cursor invisible
cursor_left	`cub1`	`le`	Mueve el cursor un caracter hacia la izquierda
cursor_mem_address	`mrcup`	`CM`	Direccionamiento relativo del cursor a través de memoria
cursor_normal	`cnorm`	`ve`	Vuelve el cursor a modo normal (deshace vs/vi)
cursor_right	`cuf1`	`nd`	Espacio no destructivo (cursor a la derecha)
cursor_to_ll	`ll`	`ll`	Última línea, primera columna (sin cup)
cursor_up	`cuu1`	`up`	Subir línea (cursor hacia arriba)
cursor_visible	`cvvis`	`vs`	Hacer el cursor muy visible
define_bit_image_region	`defbi`	`Yx`	Definir región de imagen de bits rectangular (usar tparm)
define_char	`defc`	`ZE`	Definir caracter en conjunto de caracteres
delete_character	`dch1`	`dc`	Borrar caracter (P*)
delete_line	`dl1`	`dl`	Borrar línea (P*)
device_type	`devt`	`dv`	Indica soporte de idioma/conjuto de código
dis_status_line	`dsl`	`ds`	Desactiva línea de estado
display_pc_char	`dispc`	`S1`	Imprime el caracter pc
down_half_line	`hd`	`hd`	Baja media línea (1/2 avance de línea hacia delante)
ena_acs	`enacs`	`eA`	activa conjunto de car. altern.
end_bit_image_region	`endbi`	`Yy`	Fin de región de imagen por bits (usar tparm)
enter_alt_charset_mode	`smacs`	`as`	Comienza un conjunto de caracteres alternativo (P)
enter_am_mode	`smam`	`SA`	Activa márgenes automáticos
enter_blink_mode	`blink`	`mb`	Activa caracteres intermitentes
enter_bold_mode	`bold`	`md`	Activa el modo negrita(de brillo extra)
enter_ca_mode	`smcup`	`ti`	Cadena al principio de los programas que usen cup
enter_delete_mode	`smdc`	`dm`	Modo de borrado (avtivado)
enter_dim_mode	`dim`	`mh`	Activa el modo de menor brillo
enter_doublewide_mode	`swidm`	`ZF`	Activa el modo de doble ancho
enter_draft_quality	`sdrfq`	`ZG`	Activa el modo de calidad de borrador
enter_insert_mode	`smir`	`im`	Activa el modo de inserción (activado);
enter_italics_mode	`sitm`	`ZH`	Activa el modo en cursiva
enter_leftward_mode	`slm`	`ZI`	Activa el movimiento del carro hacia la izquierda
enter_micro_mode	`smicm`	`ZJ`	Activa los atributos de micro-movimiento
enter_near_letter_quality	`snlq`	`ZK`	Activa impresión NLQ
enter_normal_quality	`snrmq`	`ZL`	Activa modo de impresión de calidad normal
enter_pc_charset_mode	`smpch`	`S2`	Activa el modo de impresión del conjunto de caracteres PC

enter_protected_mode	prot	mp	Activa el modo protegido
enter_reverse_mode	rev	mr	Activa el modo de video inverso
enter_scancode_mode	smsc	S4	Activa el modo de codigós de escaneado de PC
enter_secure_mode	invis	mk	Activa el modo vacío (caracteres invisibles)
enter_shadow_mode	sshm	ZM	Activa la impresión en modo de sombra
enter_standout_mode	smso	so	Activa el modo destacado
enter_subscript_mode	ssubm	ZN	Activa el modo de subíndice
enter_superscript_mode	ssupm	ZO	Activa el modo de superíndice
enter_underline_mode	smul	us	Comienza el modo de subrayado
enter_upward_mode	sum	ZP	Permite el movimiento hacia arriba del carro
enter_xon_mode	smxon	SX	Activa el protocolo xon/xoff
erase_chars	ech	ec	Borra #1 caracteres (PG)
exit_alt_charset_mode	rmacs	ae	Fin de conjunto de caracteres alternativo (P)
exit_am_mode	rmam	RA	Desactiva los márgenes automáticos
exit_attribute_mode	sgr0	me	Desactiva todos los atributos
exit_ca_mode	rmcup	te	Cadena para terminar los programas que usan cup
exit_delete_mode	rmdc	ed	Fin del modo de borrado
exit_doublewide_mode	rwidm	ZQ	Desactiva la impresión en doble ancho
exit_insert_mode	rmir	ei	Fin del modo de inserción
exit_italics_mode	ritm	ZR	Desactiva la impresión de cursiva
exit_leftward_mode	rlm	ZS	Activa el movimiento del carro (normal) hacia la derecha
exit_micro_mode	rmicm	ZT	Desactiva la capacidad de micro movimiento
exit_pc_charset_mode	rmpch	S3	Desactiva la impresión de caracteres PC
exit_scancode_mode	rmsc	S5	Desactiva el modo de escaneado de códigos PC
exit_shadow_mode	rshm	ZU	Deactiva la impresión en modo sombra
exit_standout_mode	rmso	se	Fin del modo destacado
exit_subscript_mode	rsubm	ZV	Desatciva la impresión de subíndices
exit_superscript_mode	rsupm	ZW	Desatciva la impresión de superíndices
exit_underline_mode	rmul	ue	Fin del modo de subrayado
exit_upward_mode	rum	ZX	Permite el movimiento del carro (normal) hacia abajo
exit_xon_mode	rmxon	RX	Desactiva el protocolo xon/xoff
flash_screen	flash	vb	Timbre visible (puede que no mueva el cursor)
form_feed	ff	ff	Expulsión de página en terminal de impresión (P*)
from_status_line	fsl	fs	Retorno desde la línea de estado
init_1string	is1	i1	Cadena de inicialización de la terminal
init_2string	is2	i2	Cadena de inicialización de la terminal
init_3string	is3	i3	Cadena de inicialización de la terminal
init_file	if	if	Nombre del fichero que contiene es
init_prog	iprog	iP	Ruta del programa de inicio
initialize_color	initc	Ic	Inicia la definición de color
initialize_pair	initp	Ip	Inicializa una pareja de colores
insert_character	ich1	ic	Añadir caracter (P)
insert_line	il1	al	Añadir una línea vacía (P*)
insert_padding	ip	ip	Añadir relleno después de caracter nuevo (p*)
key_a1	ka1	K1	Superior izquierda en el teclado numérico
key_a3	ka3	K3	Superior derecha en el teclado numérico
key_b2	kb2	K2	Centro del teclado numérico
key_backspace	kbs	kb	Enviado por el retroceso
key_beg	kbeg	1	Tecla de comienzo
key_btab	kcbt	kB	Tabulador inverso
key_c1	kc1	K4	Inferior izquierda en el teclado numérico

key_c3	`kc3`	`K5`	Inferior derecha en el teclado numérico
key_cancel	`kcan`	`2`	Tecla de cancelación
key_catab	`ktbc`	`ka`	Enviado por la tecla de borrado de tabuladores
key_clear	`kclr`	`kC`	Enviado por el borrado de pantalla o la tecla de borrado
key_close	`kclo`	`3`	Tecla de cerrado
key_command	`kcmd`	`4`	Tecla de órden
key_copy	`kcpy`	`5`	Tecla de copiado
key_create	`kcrt`	`6`	Tecla de creación
key_ctab	`kctab`	`kt`	Enviado por borrado de tabulador
key_dc	`kdch1`	`kD`	Enviado por la tecla de borrado de caracter
key_dl	`kdl1`	`kL`	Enviado por la tecla de borrado de línea
key_down	`kcud1`	`kd`	Enviado por la flecha hacia abajo
key_eic	`krmir`	`kM`	Enviado por rmir o smir en modo de inserción
key_end	`kend`	`7`	Fin
key_enter	`kent`	`8`	enter/envío
key_eol	`kel`	`kE`	Enviado por borrado hasta final de línea
key_eos	`ked`	`kS`	Enviado por borrado hasta fin de pantalla
key_exit	`kext`	`9`	Tecla de salida

key_f0	`kf0`	`k0`	Tecla de función F00	key_f32	`kf32`	`FM`	Tecla de función F32
key_f1	`kf1`	`k1`	Tecla de función F01	key_f33	`kf33`	`FN`	Tecla de función F33
key_f2	`kf2`	`k2`	Tecla de función F02	key_f34	`kf34`	`FO`	Tecla de función F34
key_f3	`kf3`	`k3`	Tecla de función F03	key_f35	`kf35`	`FP`	Tecla de función F35
key_f4	`kf4`	`k4`	Tecla de función F04	key_f36	`kf36`	`FQ`	Tecla de función F36
key_f5	`kf5`	`k5`	Tecla de función F05	key_f37	`kf37`	`FR`	Tecla de función F37
key_f6	`kf6`	`k6`	Tecla de función F06	key_f38	`kf38`	`FS`	Tecla de función F38
key_f7	`kf7`	`k7`	Tecla de función F07	key_f39	`kf39`	`FT`	Tecla de función F39
key_f8	`kf8`	`k8`	Tecla de función F08	key_f40	`kf40`	`FU`	Tecla de función F40
key_f9	`kf9`	`k9`	Tecla de función F09	key_f41	`kf41`	`FV`	Tecla de función F41
key_f10	`kf10`	`k;`	Tecla de función F10	key_f42	`kf42`	`FW`	Tecla de función F42
key_f11	`kf11`	`F1`	Tecla de función F11	key_f43	`kf43`	`FX`	Tecla de función F43
key_f12	`kf12`	`F2`	Tecla de función F12	key_f44	`kf44`	`FY`	Tecla de función F44
key_f13	`kf13`	`F3`	Tecla de función F13	key_f45	`kf45`	`FZ`	Tecla de función F45
key_f14	`kf14`	`F4`	Tecla de función F14	key_f46	`kf46`	`Fa`	Tecla de función F46
key_f15	`kf15`	`F5`	Tecla de función F15	key_f47	`kf47`	`Fb`	Tecla de función F47
key_f16	`kf16`	`F6`	Tecla de función F16	key_f48	`kf48`	`Fc`	Tecla de función F48
key_f17	`kf17`	`F7`	Tecla de función F17	key_f49	`kf49`	`Fd`	Tecla de función F49
key_f18	`kf18`	`F8`	Tecla de función F18	key_f50	`kf50`	`Fe`	Tecla de función F50
key_f19	`kf19`	`F9`	Tecla de función F19	key_f51	`kf51`	`Ff`	Tecla de función F51
key_f20	`kf20`	`FA`	Tecla de función F20	key_f52	`kf52`	`Fg`	Tecla de función F52
key_f21	`kf21`	`FB`	Tecla de función F21	key_f53	`kf53`	`Fh`	Tecla de función F53
key_f22	`kf22`	`FC`	Tecla de función F22	key_f54	`kf54`	`Fi`	Tecla de función F54
key_f23	`kf23`	`FD`	Tecla de función F23	key_f55	`kf55`	`Fj`	Tecla de función F55
key_f24	`kf24`	`FE`	Tecla de función F24	key_f56	`kf56`	`Fk`	Tecla de función F56
key_f25	`kf25`	`FF`	Tecla de función F25	key_f57	`kf57`	`Fl`	Tecla de función F57
key_f26	`kf26`	`FG`	Tecla de función F26	key_f58	`kf58`	`Fm`	Tecla de función F58
key_f27	`kf27`	`FH`	Tecla de función F27	key_f59	`kf59`	`Fn`	Tecla de función F59
key_f28	`kf28`	`FI`	Tecla de función F28	key_f60	`kf60`	`Fo`	Tecla de función F60
key_f29	`kf29`	`FJ`	Tecla de función F29	key_f61	`kf61`	`Fp`	Tecla de función F61
key_f30	`kf30`	`FK`	Tecla de función F30	key_f62	`kf62`	`Fq`	Tecla de función F62
key_f31	`kf31`	`FL`	Tecla de función F31	key_f63	`kf63`	`Fr`	Tecla de función F63

key_find	`kfnd`	`0`	Tecla de búsqueda

key_help	khlp	%1	Tecla de ayuda
key_home	khome	kh	Enviado por la tecla de Inicio
key_ic	kich1	kI	Enviado por la tecla de Inserción
key_il	kil1	kA	Enviado por insertar línea
key_left	kcub1	kl	Enviado por la flecha izquierda
key_ll	kll	kH	Enviado por la tecla home-down
key_mark	kmrk	%2	Tecla de marcar
key_message	kmsg	%3	Tecla de mensaje
key_move	kmov	%4	Tecla de movimiento
key_next	knxt	%5	Tecla "siguiente"
key_npage	knp	kN	Enviado por la tecla de página siguiente
key_open	kopn	%6	Tecla de apertura
key_options	kopt	%7	Tecla de opciones
key_ppage	kpp	kP	Enviado por la tecla de página previa
key_previous	kprv	%8	Tecla previa
key_print	kprt	%9	Tecla de impresión
key_redo	krdo	%0	Tecla de repetición
key_reference	kref	&1	Tecla de referencia
key_refresh	krfr	&2	Tecla de refresco
key_replace	krpl	&3	Tecla de reemplazamiento
key_restart	krst	&4	Tecla de reinicio
key_resume	kres	&5	Tecla de continuación
key_right	kcuf1	kr	Enviado por la tecla de flecha derecha
key_save	ksav	&6	Tecla de grabado
key_sbeg	kBEG	&9	Mayús. + tecla de comienzo
key_scancel	kCAN	&0	Mayús. + cancelación
key_scommand	kCMD	*1	Mayús. + tecla de órden
key_scopy	kCPY	*2	Mayús. + tecla de copiado
key_screate	kCRT	*3	Mayús. + tecla de creación
key_sdc	kDC	*4	Mayús. + suprimir
key_sdl	kDL	*5	Mayús. + suprimir línea
key_select	kslt	*6	Tecla de selección
key_send	kEND	*7	Mayús. + fin
key_seol	kEOL	*8	Mayús. + final de línea
key_sexit	kEXT	*9	Mayús. + salída
key_sf	kind	kF	Enviado por la tecla de avance
key_sfind	kFND	*0	Mayús. + tecla de búsqueda
key_shelp	kHLP	#1	Mayús. + tecla de ayuda
key_shome	kHOM	#2	Mayús. + inicio
key_sic	kIC	#3	Mayús. + tecla de inserción
key_sleft	kLFT	#4	Mayús. + izquierda
key_smessage	kMSG	%a	Mayús. + tecla de mensaje
key_smove	kMOV	%b	Mayús. + tecla de movimiento
key_snext	kNXT	%c	Mayús. + "siguiente"
key_soptions	kOPT	%d	Mayús. + tecla de opciones
key_sprevious	kPRV	%e	Mayús. + previo
key_sprint	kPRT	%f	Mayús. + tecla de impresión
key_sr	kri	kR	Enviado por la tecla de desplazamiento hacia atrás
key_sredo	kRDO	%g	Mayús. + tecla de repetición
key_sreplace	kRPL	%h	Mayús. + tecla de substitución
key_sright	kRIT	%i	Mayús. + derecha
key_srsume	kRES	%j	Mayús. + tecla de continuación
key_ssave	kSAV	!1	Mayús. + tecla de grabado

key_ssuspend	`kSPD`	`!2`	Mayús. + tecla de suspensión
key_stab	`khts`	`kT`	Enviado por la tecla de fijación de tabulador
key_sundo	`kUND`	`!3`	Mayús. + deshacer
key_suspend	`kspd`	`&7`	Suspensión
key_undo	`kund`	`&8`	Deshacer
key_up	`kcuu1`	`ku`	Enviado por la flecha hacia arriba
keypad_local	`rmkx`	`ke`	Salida del modo de transmisión de teclas numéricas
keypad_xmit	`smkx`	`ks`	Poner la terminal en modo de transmisión de teclas numéricas
lab_f0	`lf0`	`l0`	Etiqueta de la función f0 si no es f0
lab_f1	`lf1`	`l1`	Etiqueta de la función f1 si no es f1
lab_f2	`lf2`	`l2`	Etiqueta de la función f2 si no es f2
lab_f3	`lf3`	`l3`	Etiqueta de la función f3 si no es f3
lab_f4	`lf4`	`l4`	Etiqueta de la función f4 si no es f4
lab_f5	`lf5`	`l5`	Etiqueta de la función f5 si no es f5
lab_f6	`lf6`	`l6`	Etiqueta de la función f6 si no es f6
lab_f7	`lf7`	`l7`	Etiqueta de la función f7 si no es f7
lab_f8	`lf8`	`l8`	Etiqueta de la función f8 si no es f8
lab_f9	`lf9`	`l9`	Etiqueta de la función f9 si no es f9
lab_f10	`lf10`	`la`	Etiqueta de la función f10 si no es f10
label_on	`smln`	`LO`	Activa las etiquetas software
label_off	`rmln`	`LF`	Desactiva las etiquetas software
meta_off	`rmm`	`mo`	Desactiva el modo "meta"
meta_on	`smm`	`mm`	Activa el modo "meta" (8 bit)
micro_column_address	`mhpa`	`ZY`	Igual que column_address for micro adjustment
micro_down	`mcud1`	`ZZ`	Igual que cursor_down for micro adjustment
micro_left	`mcub1`	`Za`	Igual que cursor_left for micro adjustment
micro_right	`mcuf1`	`Zb`	Igual que cursor_right for micro adjustment
micro_row_address	`mvpa`	`Zc`	Igual que row_address for micro adjustment
micro_up	`mcuu1`	`Zd`	Igual que cursor_up for micro adjustment
newline	`nel`	`nw`	Nueva línea (equivale a cr seguido de lf)
order_of_pins	`porder`	`Ze`	Matches software buts to print-head pins
orig_colors	`oc`	`oc`	Resetea todas las parejas de color
orig_pair	`op`	`op`	Vuelve a establecer la pareja de color por defecto a su valor original
pad_char	`pad`	`pc`	Caracter de relleno (en vez del nulo)
parm_dch	`dch`	`DC`	Borra #1 caracteres (PG*)
parm_delete_line	`dl`	`DL`	Borra #1 líneas (PG*)
parm_down_cursor	`cud`	`DO`	Desplaza el cursor hacia abajo #1 líneas (PG*)
parm_down_micro	`mcud`	`Zf`	Igual que cud para micro ajustes
parm_ich	`ich`	`IC`	Añadir #1 caracteres vacíos (PG*)
parm_index	`indn`	`SF`	Avanza #1 líneas (PG)
parm_insert_line	`il`	`AL`	Añadir #1 líneas vacías (PG*)
parm_left_cursor	`cub`	`LE`	Mueve el cursor hacia la izquierda #1 espacios (PG)
parm_left_micro	`mcub`	`Zg`	Igual que cul para micro ajustes
parm_right_cursor	`cuf`	`RI`	Mueve el cursor hacia la derecha #1 espacios (PG*)
parm_right_micro	`mcuf`	`Zh`	Igual que cuf para micro ajustes
parm_rindex	`rin`	`SR`	Retrocede #1 líneas (PG)
parm_up_cursor	`cuu`	`UP`	Mueve el cursor #1 líneas hacia arriba (PG*)
parm_up_micro	`mcuu`	`Zi`	Igual que cuu para micro ajustes
pkey_key	`pfkey`	`pk`	Programa función #1 para imprimir la cadena #2
pkey_local	`pfloc`	`pl`	Programa función #1 para ejecutar la cadena #2
pkey_xmit	`pfx`	`px`	Programa función #1 para transmitir la cadena #2
pkey_plab	`pfxl`	`xl`	Programa la tecla #1 para transmitir #2 e imprimir #3
plab_norm	`pln`	`pn`	Programa la etiqueta #1 para imprimir la cadena #2

print_screen	mc0	ps	Imprime el contenido de la pantalla
prtr_non	mc5p	p0	Activa la impresora para #1 bytes
prtr_off	mc4	pf	Desacitva la impresora
prtr_on	mc5	po	Activa la impresora
repeat_char	rep	rp	Repite el caracter #1 #2 veces. (PG*)
req_for_input	rfi	RF	Petición de entrada
reset_1string	rs1	r1	Pone la terminal el modos normales.
reset_2string	rs2	r2	Pone la terminal el modos normales.
reset_3string	rs3	r3	Pone la terminal el modos normales.
reset_file	rf	rf	Nombre del fichero con la cadena de reset
restore_cursor	rc	rc	Devuelve el cursor a la posición del último sc
row_address	vpa	cv	Posición vertical absoluta (fija la fila) (PG)
save_cursor	sc	sc	Salvado del cursor (P)
scancode_escape	scesc	S7	Escape para la emulación de código de escaneado
scroll_forward	ind	sf	Avanza el texto hacia arriba (P)
scroll_reverse	ri	sr	Avanza el texto hacia abajo (P)
select_char_set	scs	Zj	Selecciona el código de caracteres
set0_des_seq	s0ds	s0	Utilizar el conjunto de códigos 0 (EUC conjunto 0, ASCII)
set1_des_seq	s1ds	s1	Utilizar el conjunto de códigos 1
set2_des_seq	s2ds	s2	Utilizar el conjunto de códigos 2
set3_des_seq	s3ds	s3	Utilizar el conjunto de códigos 3
set_a_background	setab	AB	Fijar el color del segundo plano usando una secuencia de escape ANSI
set_a_foreground	setaf	AF	Fijar el color del primer plano usando una secuencia de escape ANSI
set_attributes	sgr	sa	Definir los atributos de vídeo (PG9)
set_background	setb	Sb	Fijar el color del segundo plano
set_bottom_margin	smgb	Zk	Fijar el margen inferior en esta línea
set_bottom_margin_parm	smgbp	Zl	Fijar el margen inferior en la línea #1 o a #2 líneas del final
set_color_band	setcolor	Yz	Cambia a la cinta de color #1
set_color_pair	scp	sp	Fijar la pareja de colores
set_foreground	setf	Sf	Fijar el color del primer plano
set_left_margin	smgl	ML	Fijar el margen izquierdo en esta columna
set_left_margin_parm	smglp	Zm	Fijar el margen izquierdo (derecho) en #1 (#2)
set_lr_margin	smglr	ML	Fijar los márgenes izquierdo y derecho
set_page_length	slines	YZ	Fijar la longitud de la página en #1 líneas (usar tparm)
set_right_margin	smgr	MR	Fijar el margen derecho en esta columna
set_right_margin_parm	smgrp	Zn	Fijar el margen derecho en la columna #1
set_tab	hts	st	Fijar una parada del tabulador en esta columna en todas las filas
set_tb_margin	smgtb	MT	Fijar los márgenes superior e inferior
set_top_margin	smgt	Zo	Fijar el margen superior en esta línea
set_top_margin_parm	smgtp	Zp	Fijar el margen superior en la línea #1
set_window	wind	wi	Esta ventana está entre las líneas #1-#2 y las columnas #3-#4
start_bit_image	sbim	Zq	Comenzar la impresión de imagen de bits
start_char_set_def	scsd	Zr	Comenazar la definición de un conjunto de caracteres
stop_bit_image	rbim	Zs	Fin de impresión de imagen de bits
stop_char_set_def	rcsd	Zt	Fin de la definicion de un conjunto de caracteres
subscript_characters	subcs	Zu	Lista de caracteres que pueden ser subíndices
superscript_characters	supcs	Zv	Lista de caracteres que pueden ser superíndices
tab	ht	ta	Desplazarse hasta la siguiente parada de tabulador (en espacios de a ocho)
these_cause_cr	docr	Zw	Estos caracteres causan un CR
to_status_line	tsl	ts	Desplazarse hasta la línea de estado, columna #1
underline_char	uc	uc	Subrayar un caracter y situarse después de él
up_half_line	hu	hu	Desplazarse media línea hacia arriba (avance de 1/2 línea inverso)

xoff_character	`xoffc`	`XF`	caracter XON
xon_character	`xonc`	`XN`	caracter XOFF

(Los siguientes atributos de cadena están presentes en la estructura term del SYSVr, aunque no están documentados en la página de manual. Los comentarios están sacados de fichero de cabecera que define la estructura term.)

label_format	`fln`	`Lf`	??
set_clock	`sclk`	`SC`	Fija el reloj
display_clock	`dclk`	`DK`	Imprime el reloj
remove_clock	`rmclk`	`RC`	Borra el reloj??
create_window	`cwin`	`CW`	Define que la ventana #1 va de #2,#3 a #4,#5
goto_window	`wingo`	`WG`	Ir a la ventana #1
hangup	`hup`	`HU`	Colgar el teléfono
dial_phone	`dial`	`DI`	Marcar el teléfono #1
quick_dial	`qdial`	`QD`	Marcar el teléfono #1, sin detectar como va la llamada
tone	`tone`	`TO`	Elegir modo de marcado por tonos
pulse	`pulse`	`PU`	Elegir modo de marcado por pulsos
flash_hook	`hook`	`fh`	Pulsar rápidamente el interruptor de colgado
fixed_pause	`pause`	`PA`	Pausa de 2-3 segundos
wait_tone	`wait`	`WA`	Esperar el tono de marcado
user0	`u0`	`u0`	Cadena de usuario # 0
user1	`u1`	`u1`	Cadena de usuario # 1
user2	`u2`	`u2`	Cadena de usuario # 2
user3	`u3`	`u3`	Cadena de usuario # 3
user4	`u4`	`u4`	Cadena de usuario # 4
user5	`u5`	`u5`	Cadena de usuario # 5
user6	`u6`	`u6`	Cadena de usuario # 6
user7	`u7`	`u7`	Cadena de usuario # 7
user8	`u8`	`u8`	Cadena de usuario # 8
user9	`u9`	`u9`	Cadena de usuario # 9
get_mouse	`getm`	`Gm`	Curses debería responder a los mensajes de botones
key_mouse	`kmous`	`Km`	??
mouse_info	`minfo`	`Mi`	Información del estado del ratón
pc_term_options	`pctrm`	`S6`	Opciones de terminal del PC
req_mouse_pos	`reqmp`	`RQ`	Petición de la posición del ratón
zero_motion	`zerom`	`Zx`	No desplazarse al detectar el siguiente caracter

8.23 Esquema de las Funciones de [N]Curses

A continuación se puede ver un resumen de los diferentes paquetes (n)curses. La primera columna corresponde a la curses de bsd (que forma parte del slackware 2.1.0 y Sun-OS 4.x), en la segunda tenemos la curses del sysv (en Sun-OS 5.4 /Solaris 2) y la tercera es ncurses (versión 1.8.6).

En la cuarta columna se encuentra un referencia a la página en la que se describe la función (si es que se describe en algún lado).

x el paquete tiene esta función.

n la función no ha sido implementada aún.

Funcion	BSD	SYSV	Nc.	Pag.
`_init_trace()`			x	121
`_traceattr(mode)`			x	121
`_tracef(char *, ...)`			x	121
`addbytes(...)`	x			
`addch(ch)`	x	x	x	98
`addchnstr(...)`		x	x	98
`addchstr(chstr)`		x	x	98
`addnstr(...)`		x	x	98
`addnwstr(...)`		x		
`addstr(str)`	x	x	x	98
`addwch(...)`		x		
`addwchnstr(...)`		x		
`addwchstr(...)`		x		
`addwstr(...)`		x		
`adjcurspos()`		x		
`attroff(attr)`		x	x	113
`attron(attr)`		x	x	113
`attrset(attr)`		x	x	113
`baudrate()`	x	x	x	106
`beep()`		x	x	118
`bkgd(ch)`		x	x	102
`bkgdset(ch)`		x	x	102
`border(...)`		x	x	100
`box(...)`	x	x	x	100
`can_change_color()`		x	x	113
`cbreak()`	x	x	x	105
`clear()`	x	x	x	110
`clearok(...)`	x	x	x	104
`clrtobot()`	x	x	x	110
`clrtoeol()`	x	x	x	110
`color_content(...)`		x	x	114
`copywin(...)`		x	x	98
`crmode()`	x	x	x	105
`curs_set(bf)`		x	x	115
`curserr()`		x		
`def_prog_mode()`		x	x	119
`def_shell_mode()`		x	x	119
`del_curterm(...)`		x	x	120
`delay_output(ms)`		x	x	119
`delch()`	x	x	x	100
`deleteln()`	x	x	x	100
`delscreen(...)`		x	x,n	95
`delwin(win)`	x	x	x	97
`derwin(...)`		x	x	97
`doupdate()`		x	x	110
`drainio(int)`		x		
`dupwin(win)`		x	x	97
`echo()`	x	x	x	105
`echochar(ch)`		x	x	99
`echowchar(ch)`		x		
`endwin()`	x	x	x	95
`erase()`	x	x	x	109
`erasechar()`	x	x	x	106
`filter()`		x	x	119
`flash()`		x	x	118
`flushinp()`		x	x	119
`flushok(...)`	x			
`garbagedlines(...)`		x		
`garbagedwin(win)`		x		
`getattrs(win)`		x	x	113
`getbegyx(...)`		x	x	116
`getbkgd(win)`		x		
`getbmap()`		x		
`getcap(str)`	x			
`getch()`	x	x	x	102
`getmaxx(win)`		x	x	116
`getmaxy(win)`		x	x	116
`getmaxyx(...)`		x	x	116
`getmouse()`		x		
`getnwstr(...)`		x		
`getparyx(...)`		x	x	115
`getstr(str)`	x	x	x	102
`getsyx(...)`		x	x	116
`gettmode()`	x	x		
`getwch(...)`		x		
`getwin(...)`		x		
`getwin(FILE *)`		x	x,n	119
`getwstr(...)`		x		
`getyx(...)`	x	x	x	115
`halfdelay(t)`		x	x	105
`has_colors()`		x	x	113
`has_ic()`		x	x,n	106
`has_il()`		x	x,n	106
`hline(...)`		x	x	100
`idcok(...)`		x	x,n	104
`idlok(...)`	x	x	x	104
`immedok(...)`		x	x	104
`inch()`	x	x	x	103
`inchnstr(...)`		x	x,n	103
`inchstr(...)`		x	x,n	103
`init_color(...)`		x	x	114
`init_pair(...)`		x	x	113
`initscr()`	x	x	x	94
`innstr(...)`		x	x,n	103
`innwstr(...)`		x		
`insch(c)`	x	x	x	99
`insdelln(n)`		x	x	99
`insertln()`	x	x	x	99
`insnstr(...)`		x	x	99
`insstr(str)`		x	x	99
`instr(str)`		x	x,n	103
`inswch(...)`		x		
`inswstr(...)`		x		
`intrflush(...)`		x	x	106
`inwch(...)`		x		
`inwchnstr(...)`		x		
`inwchstr(...)`		x		
`inwchstr(...)`		x		
`inwstr(...)`		x		
`is_linetouched(...)`		x	x	111
`is_wintouched(win)`		x	x	111
`isendwin()`		x	x	95
`keyname(c)`		x	x	119
`keypad(...)`		x	x	104
`killchar()`	x	x	x	106
`leaveok(...)`	x	x	x	104
`longname()`	x	x	x	106
`map_button(long)`		x		
`meta(...)`		x	x	104
`mouse_off(long)`		x		
`mouse_on(long)`		x		
`mouse_set(long)`		x		
`move(...)`	x	x	x	115
`movenextch()`		x		
`moveprevch()`		x		
`mvaddbytes(...)`	x			
`mvaddch(...)`	x	x	x	98
`mvaddchnstr(...)`		x	x	98
`mvaddchstr(...)`		x	x	98
`mvaddnstr(...)`		x	x	98
`mvaddnwstr(...)`		x		

mvaddstr(...)	x	x	x	98
mvaddwch(...)		x		
mvaddwchnstr(...)		x		
mvaddwchstr(...)		x		
mvaddwstr(...)		x		
mvcur(...)	x	x	x	121
mvdelch(...)	x	x	x	100
mvderwin(...)		x	x,n	97
mvgetch(...)	x	x	x	102
mvgetnwstr(...)		x		
mvgetstr(...)	x	x	x	102
mvgetwch(...)		x		
mvgetwstr(...)		x		
mvhline(...)		x		
mvinch(...)	x	x	x	103
mvinchnstr(...)		x	x,n	103
mvinchstr(...)		x	x,n	103
mvinnstr(...)		x	x,n	103
mvinnwstr(...)		x		
mvinsch(...)	x	x	x	99
mvinsnstr(...)		x	x	100
mvinsnwstr(...)		x		
mvinsstr(...)		x	x	100
mvinstr(...)		x	x,n	103
mvinswch(...)		x		
mvinswstr(...)		x		
mvinwch(...)		x		
mvinwchnstr(...)		x		
mvinwchstr(...)		x		
mvinwstr(...)		x		
mvprintw(...)	x	x	x	99
mvscanw(...)	x	x	x	103
mvvline(...)		x		
mvwaddbytes(...)	x			
mvwaddch(...)	x	x	x	98
mvwaddchnstr(...)		x	x	98
mvwaddchstr(...)		x	x	98
mvwaddnstr(...)		x	x	98
mvwaddnwstr(...)		x		
mvwaddstr(...)	x	x	x	98
mvwaddwch(...)		x		
mvwaddwchnstr(...)		x		
mvwaddwchstr(...)		x		
mvwaddwstr(...)		x		
mvwdelch(...)	x	x	x	100
mvwgetch(...)	x	x	x	102
mvwgetnwstr(...)		x		
mvwgetstr(...)	x	x	x	102
mvwgetwch(...)		x		
mvwgetwstr(...)		x		
mvwhline(...)		x		
mvwin(...)	x	x	x	97
mvwinch(...)	x	x	x	103
mvwinchnstr(...)		x	x,n	103
mvwinchstr(...)		x	x,n	103
mvwinnstr(...)		x	x,n	103
mvwinnwstr(...)		x		
mvwinsch(...)	x	x	x	99
mvwinsnstr(...)		x	x	100
mvwinsstr(...)		x	x	100
mvwinstr(...)		x	x,n	103
mvwinswch(...)		x		
mvwinswstr(...)		x		
mvwinwch(...)		x		
mvwinwchnstr(...)		x		
mvwinwchstr(...)		x		
mvwinwstr(...)		x		
mvwprintw(...)	x	x	x	99
mvwscanw(...)	x	x	x	103
mvwvline(...)		x		
napms(ms)		x	x	119
newkey(...)		x		
newpad(...)		x	x	117
newscreen(...)		x		
newterm(...)		x	x	94
newwin(...)	x	x	x	95
nl()	x	x	x	104
nocbreak()	x	x	x	105
nocrmode()	x	x	x	105
nodelay(...)		x	x	105
noecho()	x	x	x	105
nonl()	x	x	x	104
noqiflush()		x	x,n	106
noraw()	x	x	x	105
notimeout(...)		x	x	106
overlay(...)	x	x	x	97
overwrite(...)	x	x	x	97
pair_content(...)		x	x	114
pechochar(...)		x	x	118
pechowchar(...)		x		
pnoutrefresh(...)		x	x	118
prefresh(...)		x	x	118
printw(...)	x	x	x	99
putp(char *)		x	x	121
putwin(...)		x	x,n	119
qiflush()		x	x,n	106
raw()	x	x	x	105
redrawwin(win)		x	x	111
refresh()	x	x	x	110
request_mouse_pos()		x		
reset_prog_mode()		x	x	119
reset_shell_mode()		x	x	119
resetty()	x	x	x	119
restartterm(...)		x	x,n	120
ripoffline(...)		x	x	119
savetty()	x	x	x	119
scanw(...)	x	x	x	103
scr_dump(char *)		x	x,n	120
scr_init(char *)		x	x,n	120
scr_restore(char *)		x	x,n	120
scr_set(char *)		x	x,n	120
scrl(n)		x	x	116
scroll(win)	x	x	x	116
scrollok(...)	x	x	x	116
set_curterm(...)		x	x	120
set_term(...)		x	x	95
setcurscreen(SCREEN *)		x		
setscrreg(...)		x	x	116
setsyx(...)		x	x	116
setterm(char *)	x	x	x	120
setupterm(...)		x	x	120
slk_attroff(attr)		x	x,n	118
slk_attron(attr)		x	x,n	118
slk_attrset(attr)		x	x,n	118
slk_clear()		x	x	118
slk_init(fmt)		x	x	118
slk_label(labnum)		x	x	118
slk_noutrefresh()		x	x	118
slk_refresh()		x	x	118
slk_restore()		x	x	118
slk_set(...)		x	x	118
slk_touch()		x	x	118

`standend()`	x	x	x	113
`standout()`	x	x	x	113
`start_color()`		x	x	113
`subpad(...)`		x	x	117
`subwin(...)`	x	x	x	97
`syncok(...)`		x	x,n	97
`termattrs()`		x	x,n	106
`termname()`		x	x,n	107
`tgetent(...)`		x	x	120
`tgetflag(char [2])`		x	x	120
`tgetnum(char [2])`		x	x	120
`tgetstr(...)`		x	x	120
`tgoto(...)`		x	x	120
`tigetflag(...)`		x	x	121
`tigetnum(...)`		x	x	121
`tigetstr(...)`		x	x	121
`timeout(t)`		x	x	105
`touchline(...)`	x	x	x	111
`touchwin(win)`	x	x	x	111
`tparm(...)`		x	x	121
`tputs(...)`			x	120
`traceoff()`		x	x	121
`traceon()`		x	x	121
`typeahead(fd)`		x	x	106
`unctrl(chtype c)`		x	x	119
`ungetch(ch)`		x	x	102
`ungetwch(c)`		x		
`untouchwin(win)`		x	x	111
`use_env(bf)`		x	x	119
`vidattr(...)`		x	x	121
`vidputs(...)`		x	x	121
`vidupdate(...)`		x		
`vline(...)`		x	x	100
`vwprintw(...)`		x	x	99
`vwscanw(...)`		x	x	103
`waddbytes(...)`	x			
`waddch(...)`	x	x	x	98
`waddchnstr(...)`		x	x	98
`waddchstr(...)`		x	x	98
`waddnstr(...)`		x	x	98
`waddnwstr(...)`		x		
`waddstr(...)`	x	x	x	98
`waddwch(...)`		x		
`waddwchnstr(...)`		x		
`waddwchstr(...)`		x		
`waddwstr(...)`		x		
`wadjcurspos(win)`		x		
`wattroff(...)`		x	x	113
`wattron(...)`		x	x	113
`wattrset(...)`		x	x	113
`wbkgd(...)`		x	x	102
`wbkgdset(...)`		x	x	102
`wborder(...)`		x	x	100
`wclear(win)`	x	x	x	110
`wclrtobot(win)`	x	x	x	110
`wclrtoeol(win)`	x	x	x	110
`wcursyncup(win)`		x	x,n	97
`wdelch(win)`	x	x	x	100
`wdeleteln(win)`	x	x	x	100
`wechochar(...)`		x	x	99
`wechowchar(...)`		x		
`werase(win)`	x	x	x	109
`wgetch(win)`	x	x	x	102
`wgetnstr(...)`		x	x	102
`wgetnwstr(...)`		x		
`wgetstr(...)`	x	x	x	102
`wgetwch(...)`		x		
`wgetwstr(...)`		x		
`whline()`		x		
`whline(...)`		x		
`whline(...)`		x	x	100
`winch(win)`	x	x	x	103
`winchnstr(...)`		x	x,n	103
`winchstr(...)`		x	x,n	103
`winnstr(...)`		x	x,n	103
`winnwstr(...)`		x		
`winsch(...)`	x	x	x	99
`winsdelln(...)`	x	x	x	99
`winsertln(win)`		x	x	99
`winsnstr(...)`		x	x	100
`winsnwstr(...)`		x		
`winsstr(...)`		x	x	100
`winstr(...)`		x	x,n	103
`winswch(...)`		x		
`winswstr(...)`		x		
`winwch(...)`		x		
`winwchnstr(...)`		x		
`winwchstr(...)`		x		
`winwstr(...)`		x		
`wmouse_position(...)`		x		
`wmove(...)`	x	x	x	115
`wmovenextch(win)`		x		
`wmoveprevch(win)`		x		
`wnoutrefresh(win)`		x	x	110
`wprintw(...)`	x	x	x	99
`wredrawln(...)`		x	x	111
`wrefresh(win)`	x	x	x	110
`wscanw(...)`	x	x	x	103
`wscrl(...)`		x	x	116
`wsetscrreg(...)`		x	x	116
`wstandend(win)`	x	x	x	113
`wstandout(win)`	x	x	x	113
`wsyncdown(win)`		x	x,n	97
`wsyncup(win)`		x	x,n	97
`wtimeout(...)`		x	x	105
`wtouchln(...)`		x	x	111
`wvline()`		x		
`wvline(...)`		x		
`wvline(...)`		x	x	100

Continuará...
Sven Goldt Guía del Programador de Linux

Capítulo 9

Programación de los Puertos de E/S

Normalmente, un PC tiene al menos dos interfaces serie y una paralelo. Estas interfaces son dispositivos especiales y se mapean como sigue:

- $/dev/ttyS0 - /dev/ttyS\mathbf{n}$
  estos son los dispositivos serie RS232 0-**n** donde **n** depende de su hardware.
- $/dev/cua0 - /dev/cua\mathbf{n}$
  estos son los dispositivos RS232 0-**n** donde **n** depende de su hardware.
- $/dev/lp0 - /dev/lp\mathbf{n}$
  estos son los dispositivos paralelos 0-**n** donde **n** depende de su hardware.
- $/dev/js0 - /dev/js\mathbf{n}$
  estos son los dispositivos de joystick 0-**n** donde $0 <= \mathbf{n} <= 1$.

La diferencia entre los dispositivos $/dev/ttyS*$ y $/dev/cua*$ consiste en como se maneja la llamada a open(2). Se supone que los dispositivos $/dev/cua*$ se deben usar como dispositivos de llamada saliente y por lo tanto, al invocar a open(2), reciben parámetros por defecto diferentes a los que reciben los dispositivos $/dev/ttyS*$, que se inicializan para llamadas entrantes y salientes. Por defecto los dispositivos son dispositivos controladores para aquellos procesos que los abrieron. Normalmente estos dispositivos especiales deberían manejarse con peticiones ioctl(), pero POSIX prefirió definir nuevas funciones para manejar los terminales asíncronos que dependen fuertemente de la estructura termios. Ambos métodos requieren que se incluya $< termios.h >$.

1. método ioctl:
   TCSBRK, TCSBRKP, TCGETA (obtener atributos), TCSETA (poner atributos)
   Peticiones de control de E/S de Terminal (TIOC):
   TIOCGSOFTCAR (obtener portadora soft), TIOCSSOFTCAR (poner portadora soft), TIOCSCTTY (poner tty controlador), TIOCMGET (obtener líneas de módem), TIOCMSET (activar líneas de

módem), TIOCGSERIAL, TIOCSSERIAL, TIOCSERCONFIG, TIOCSERGWILD, TIOCSERSWILD, TIOCSERGSTRUCT, TIOCMBIS, TIOCMBIC, ...

2. método POSIX:
   tcgetattr(), tcsetattr(), tcsendbreak(), tcdrain(), tcflush(), tcflow(), tcgetpgrp(), tcsetpgrp()
   cfsetispeed(), cfgetispeed(), cfsetospeed(), cfgetospeed()

3. otros métodos:
   outb,inb para la programación a bajo nivel, como por ejemplo para usar el puerto de la impresora con algo que no sea una impresora.

9.1 Programación del Ratón

Los ratones se conectan o bien a un puerto serie o bien directamente al bus AT. Diferentes tipos de ratones envían diferentes tipos de datos, lo que hace la programación del ratón algo más complicada aún. Pero Andrew Haylett fue tan amable que puso un copyright generoso en su programa selection, lo que significa que puede usar estas rutinas de ratón para sus propios programas. Junto con este manual puede encontrar la versión ***release** previa de selection-1.8[1] junto con la nota de COPYRIGHT. Por otra parte, X11 ofrece una API cómoda de usar, así que las rutinas de Andrew se deberían usar únicamente para aplicaciones que no sean X11. Sólo son necesarios los módulos mouse.c y mouse.h del paquete selection. Para recibir los eventos del ratón básicamente hay que llamar a ms_init() y get_ms_event(). ms_init necesita los 10 parámetos siguientes:

1. *int acceleration*
   es el factor de aceleración. Si mueve el ratón más de *delta* pixels, la velocidad de movimiento se incrementa dependiendo de este valor.

2. *int baud*
   es la velocidad en bps que usa su ratón (normalmente 1200).

3. *int delta*
   éste es el número de pixels que debe mover el ratón antes de que comience la aceleración.

4. *char *device*
   es el nombre de su dispositivo de ratón (por ejemplo /dev/mouse)

5. *int toggle*
   conmuta la línea de módem del ratón DTR, RTS o ambas durante la inicialización (normalmente 0).

6. *int sample*
   la resolución (en dpi) de su ratón (normalmente 100).

7. *mouse_type mouse*
   el identificador del ratón conectado, como P_MSC (Mouse Systems Corp.) para mi ratón ;).

8. *int slack*
   cantidad de elasticidad para el "salto circular"[2], lo que significa que si *slack* es -1, un intento de mover el ratón más allá del borde de la pantalla dejará el cursor en el borde. Los valores >= 0 significan que el cursor del ratón pasará al otro extremo de la pantalla tras mover el ratón *slack* pixels contra el borde.

[1] N. del T.: en el momento de traducir esto había una versión mas reciente disponible

[2] N. del T.: traducción libre de *wraparound*, que es sinónimo de *word wrapping* y se refiere al salto automático al otro extremo de la pantalla cuando algo no cabe en un lado de la misma

9. *int maxx*
   la resolución de su terminal actual en la dirección x. Con el tipo de letra por defecto, un carácter tiene una anchura de 10 pixels y por lo tanto la resolución total de la pantalla en x es 10*80-1.

10. *int maxy*
    la resolución de su terminal actual en la dirección y. Con el tipo de letra por defecto, un carácter tiene una altura de 12 pixels y por lo tanto la resolución total de la pantalla en y es 12*25-1.

get_ms_event() necesita únicamente un puntero a una estructura ms_event. Si ocurre un error, get_ms_event() devuelve -1. Cuando todo va bien, devuelve 0 y la estructura ms_event contiene el estado actual del ratón.

9.2 Programación del Módem

Véase el ejemplo miniterm.c
Usar termios para controlar el puerto RS232.
Usar los comandos Hayes para controlar el módem.

9.3 Programación de la Impresora

Véase el ejemplo checklp.c
No usar termios para controlar el puerto de la impresora. Usar ioctl e inb/outb si fuera necesario.
Usar comandos Epson, Postscript, PCL, etc. para controlar la impresora.
$< linux/lp.h >$
llamadas ioctl: LPCHAR, LPTIME, LPABORT, LPSETIRQ, LPGETIRQ, LP-WAIT
inb/outb para estado y control del puerto.

9.4 Programación del Joystick

Véase ejemplo js.c en el paquete del módulo cargable del núcleo para el joystick.
$< linux/joystick.h >$
llamadas
ioctl: JS_SET_CAL, JS_GET_CAL, JS_SET_TIMEOUT, JS_GET_TIMEOUT, JS_SET_TIMELIMIT, JS_GET_TIMELIMIT, JS_GET_ALL, JS_SET_ALL. Una lectura en /dev/js**n** devolverá la estructura JS_DATA_TYPE.

Capítulo 10

Conversión de Aplicaciones a Linux

Matt Welsh
mdw@cs.cornell.edu 26 de Enero de 1995

10.1 Introducción

La conversión de aplicaciones UNIX al sistema operativo Linux es extremadamente fácil. Linux, y la biblioteca GNU C usada por él, han sido diseñados con la portabilidad de las aplicaciones en mente, lo que significa que muchas aplicaciones compilarán con solo ejecutar `make`. Aquellas que no lo hagan, generalmente usarán alguna característica oscura de una implementación particular, o dependerán fuertemente del comportamiento indocumentado o indefinido de, por ejemplo, una llamada particular del sistema.

Linux obedece casi completamente el estándar IEEE 1003.1-1988 (POSIX.1), pero no ha sido certificado como tal. De igual forma, Linux también implementa muchas de las características que se encuentran en las variantes SVID y BSD de UNIX, pero de nuevo no se adhiere a ellas necesariamente en todos los casos. En general, Linux ha sido diseñado para ser compatible con otras implementaciones de UNIX, para hacer la conversión de aplicaciones más fácil, y en ciertas ocasiones ha mejorado o corregido el comportamiento encontrado en esas implementaciones.

Como ejemplo, el argumento *timeout* que se le pasa a la llamada del sistema *select()* es realmente decrementado por Linux durante la operación de sondeo. Otras implementaciones no modifican este valor para nada, y aquellas aplicaciones que no esperen esto pueden dejar de funcionar cuando se compilen bajo Linux. Las páginas del manual de BSD y SunOS para *select()* avisan de que en una "implementación futura", la llamada del sistema puede modificar el puntero *timeout*. Desgraciadamente, muchas aplicaciones todavía presuponen que el valor permanecerá intacto.

El objetivo de este artículo es proporcionar una vista general de los principales asuntos asociados a la conversión de aplicaciones a Linux, resaltando las diferencias entre Linux, POSIX.1, SVID y BSD en las siguientes áreas: gestión de señales, E/S de terminales, control de procesos y obtención de información y compilación portable condicional.

10.2 Gestión de Señales

A lo largo de los años la definición y semántica de las señales han sido modificadas de diferentes formas por diferentes implementaciones de UNIX. Hoy en día hay dos clases principales de símbolos: *no fiables* y *fiables*. Las señales no fiables son aquellas para las cuales el gestor de la señal no continúa instalado una vez llamado. Esta señales "mono-disparo" deben reinstalar el gestor de la señal dentro del propio gestor de la señal, si el programa desea que la señal siga instalada. A causa de esto, existe una condición de carrera en la cual la señal puede llegar de nuevo antes de que el gestor este reinstalado, lo que puede hacer que, o bien la señal se pierda, o bien que se dispare el comportamiento original de la señal (tal como matar el proceso). Por lo tanto, estas señales son "no fiables" puesto que la captura de la señal y la operación de reinstalación del gestor no son atómicas.

Con la semántica de las señales no fiables, las llamadas del sistema no son reiniciadas automaticamente cuando son interrumpidas por una señal. Por lo tanto, para que un programa tenga en cuenta todas las posibilidades, es necesario que el programa compruebe el valor de *errno* tras cada llamada del sistema, y reejecute la llamada si su valor es *EINTR*.

De forma similar, la semántica de las señales no fiables no proporciona una forma fácil de obtener una operación de pausa atómica (poner un proceso a dormir hasta que llegue una señal). A causa de la naturaleza no fiable de la reinstalación de los gestores de señales, hay casos en los cuales la señal puede llegar sin que el programa se dé cuenta de ello.

Por otro lado, con la semántica de las señales fiables, el gestor de la señal permanece instalado una vez llamado, y se evita la condición de carrera. También, ciertas llamadas del sistema puede ser reiniciadas y es posible hacer una operación de pausa atómica por medio de la funcion POSIX *sigsuspend*.

10.2.1 Señales en SVR4, BSD, y POSIX.1

La implementación de señales SVR4 incorpora las funciones *signal*, *sigset*, *sighold*, *sigrelse*, *sigignore* y *sigpause*. La función *signal* bajo SVR4 es idéntica a las clásicas señales UNIX V7, y proporciona únicamente señales no fiables. Las otras funciones sí proporcionan señales con reinstalación automática del gestor de la señal, pero no se soporta el reiniciado de las señales del sistema.

Bajo BSD, se soportan las funciones *signal*, *sigvec*, *sigblock*, *sigsetmask* y *sigpause*. Todas las funciones proporcionan señales fiables con reiniciado de las llamadas del sistema por defecto, pero dicho comportamiento puede ser inhabilitado a voluntad por el programador.

Bajo POSIX.1 se proporcionan las funciones *sigaction*, *sigprocmask*, *sigpending*, y *sigsuspend*. Nótese que no existe la funcion *signal* y que, de acuerdo con POSIX.1, debe despreciarse. Estas funciones proporcionan señales fiables, pero no se define el comportamiento de las llamadas del sistema. Si se usa *sigaction* bajo SVR4 y BSD, el reiniciado de las llamadas del sistema está deshabilitado por defecto, pero puede activarse si se especifica el flag de señal `SA_RESTART`.

Por lo tanto, la "mejor" forma de usar las señales en un programa es usar *sigaction*, que permite especificar explícitamente el comportamiento de los gestores de señales. Sin embargo, todavía hay muchas aplicaciones que usan *signal*, y como podemos ver arriba, *signal* proporciona semánticas diferentes bajo SV4R y BSD.

10.2.2 Opciones de Señales en Linux

En Linux se definen los siguiente valores para el miembro `sa_flags` de la estructura `sigaction`.

- `SA_NOCLDSTOP`: No enviar `SIGCHLD` cuando un proceso hijo se detiene.
- `SA_RESTART`: Forzar el reiniciado de ciertas llamadas del sistema cuando sean interrumpidan por un gestor de señal.
- `SA_NOMASK`: Deshabilitar la máscara de señales (que bloquea las señales durante la ejecución de un gestor de señales).
- `SA_ONESHOT`: Eliminar el gestor de señal tras la ejecución. Nótese que SVR4 usa `SA_RESETHAND` para indicar lo mismo.
- `SA_INTERRUPT`: Definido en Linux, pero no usado. Bajo SunOS, las llamadas del sistema se reiniciaban automáticamente, y este flag inhabilitaba ese comportamiento.
- `SA_STACK`: Actualmente una operación nula, a ser usada por las pilas de señales.

Nótese que POSIX.1 define únicamente `SA_NOCLDSTOP`, y que hay varias opciones más definidas por SVR4 que no están disponibles en Linux. Cuando se porten aplicaciones que usen *sigaction*, puede que necesite modificar los valores de `sa_flags` para obtener el comportamiento apropiado.

10.2.3 *signal* en Linux

En Linux, la función *signal* es equivalente a usar *sigaction* con las opciones `SA_ONESHOT` y `SA_NOMASK`. Esto es, corresponde a la semántica clásica de señales no fiables usada en SVR4.

Si desea que *signal* use la semántica de BSD, la mayoría de los sistemas Linux proporcionan una biblioteca de compatibilidad BSD con la cual se puede enlazar. Para usar esta biblioteca, debería añadir las opciones

```
-I/usr/include/bsd -lbsd
```

a la línea de ordenes de compilación. Cuando porte aplicaciones que usen *signal*, preste mucha atención a las suposiciones que hace el programa sobre los gestores de señales y modifique el codigo (o compile con las definiciones apropiadas) para obtener el comportamiento adecuado.

10.2.4 Señales soportadas por Linux

Linux soporta casi todas las señales proporcionadas por SVR4, BSD y POSIX, con algunas excepciones:

- `SIGEMT` no está soportada. Corresponde a un fallo de hardware en SVR4 y BSD.
- `SIGINFO` no está soportada. Se usa para peticiones de información del teclado en SVR4.
- `SIGSYS` no está soportada. Se refiere a una llamada del sistema no válida en SVR4 y BSD. Si enlaza con `libbsd`, esta señal se redefine como `SIGUNUSED`.
- `SIGABRT` y `SIGIOT` son idénticas.
- `SIGIO`, `SIGPOLL`, y `SIGURG` son idénticas.
- `SIGBUS` se define como `SIGUNUSED`. Técnicamente no existe un "error de bus" en el entorno Linux.

10.3 E/S de Terminal

Al igual que ocurre con las señales, el control de las E/S de terminales tiene tres implementaciones diferentes: SVR4, BSD y POSIX.1.

SVR4 usa la estructura `termio` y varias llamadas *ioctl* (tales como `TCSETA`, `TCGETA`, etc.) con un dispositivo de terminal, para obtener y fijar los parámetros con la estructura `termio`. Esta estructura tiene la siguiente forma:

```
struct termio {
  unsigned short c_iflag;  /* Modos de Entrada */
  unsigned short c_oflag;  /* Modos de Salida */
  unsigned short c_cflag;  /* Modos de Control*/
  unsigned short c_lflag;  /* Modos de Disciplina de L{\'\i}nea */
  char c_line;             /* Disciplina de L{\'\i}nea */
  unsigned char c_cc[NCC]; /* Caracteres de Control */
};
```

En BSD, se usa la estructura `sgtty` junto con varias llamadas *ioctl*, tales como `TIOCGETP`, `TIOCSETP`, etc.

En POSIX, se usa la estructura `termios`, junto con varias funciones definidas por POSIX.1, tales como *tcsetattr* and *tcgetattr*. La estructura `termios` es idéntica a la estructura `struct termio` usada por SVR4, pero los tipos están renombrados (como `tcflag_t` en vez de `unsigned short`) y se usa `NCCS` para el tamaño del array `c_cc`.

En Linux, el nucleo soporta directamente tanto POSIX.1 *termios* como SVR4 *termio*. Esto significa que si su programa usa uno de estos dos métodos para acceder a las E/S de terminal, debería compilar directamente en Linux.

Si alguna vez está en duda, es fácil modificar el código que use `termio` para usar `termios`, usando un pequeño conocimiento de ambos métodos. Por suerte esto nunca debería ser necesario. Pero, si un programa intenta usar el campo `c_line` de la estructura `termio`, preste especial atención. Para casi todas las aplicaciones, este campo debería ser `N_TTY`, y si el programa presupone que está disponible algún otro tipo de disciplina, puede que tenga problemas.

Si su programa usa la implementación BSD *sgtty*, puede enlazar con `libbsd.a` como se ha indicado anteriormente. Esto proporciona un sustituto de *ioctl* que reenvía las peticiones de E/S de terminal en términos de las llamadas POSIX `termios` que usa el núcleo. Cuando compile este tipo de programas, si hay símbolos indefinidos tales como `TIOCGETP`, entonces necesitará enlazar con `libbsd`.

10.4 Control e Información de Procesos

Los programas como *ps*, *top* y *free* deben ser capaces de obtener información del núcleo sobre los procesos y recursos del sistema. De forma similar, los depuradores y herramientas similares necesitan ser capaces de controlar e inspeccionar un proceso en ejecución. Diferentes versiones de UNIX han proporcionado estas características a través de interfaces diferentes, y casi todas ellas son o bien dependientes de la máquina o bien están ligadas a un diseño particular del núcleo. Hasta ahora no ha habido una interfaz aceptada universalmente para este tipo de interacción núcleo-proceso.

10.4.1 Rutinas *kvm*

Muchos sistemas usan rutinas tales como *kvm_open*, *kvm_nlist* y *kvm_read* para acceder directamente a las estructuras de datos del núcleo a través del dispositivo */dev/kmem*. En general estos programas abrirán */dev/kmem*, leerán la tabla de símbolos del núcleo, localizarán los datos del núcleo en ejecución con esta tabla y leerán las direcciones apropiadas en el espacio de direcciones del núcleo con estas rutinas. Puesto que esto requiere que el programa del usuario y el núcleo se pongan de acuerdo en cuanto al tamaño y formato de las estructuras leídas de esta forma, tales programas deben ser reconstruidos para cada nueva revisión del núcleo, tipo de CPU, etc.

10.4.2 *ptrace* y el sistema de ficheros */proc*

La llamada del sistema *ptrace* se usa en 4.3BSD y SVID para controlar un proceso y leer información sobre él. Normalmente la usan los depuradores para, por ejemplo, detener la ejecución de un proceso en marcha y examinar su estado. En SVR4, *ptrace* ha sido sustituida por el sistema de ficheros */proc*, que se muestra como un directorio que contiene una única entrada de fichero por cada proceso en ejecución y cuyo nombre es el ID del proceso. El programa del usuario puede abrir el fichero correspondiente al proceso que le interesa y generar varias llamadas *ioctl* sobre él para controlar su ejecución u obtener información del núcleo sobre el proceso. De forma similar, el programa puede leer o escribir datos directamente en el espacio de direcciones del proceso a través del descriptor de fichero del sistema de ficheros */proc*.

10.4.3 Control de Procesos en Linux

En Linux se soporta la llamada del sistema *ptrace* para el control de los procesos, y funciona como en 4.3BSD. Linux también proporciona el sistema de ficheros */proc* para obtener información de los procesos y el sistema, pero con una semántica muy diferente. En Linux, */proc* consta de una serie de ficheros que proporcionan información general del sistema tales como uso de memoria, media de carga, estadísticas de los módulos cargados y estadísticas de la red. Se puede acceder a estos ficheros usando *read* y *write* y se puede analizar su contenido con *scanf*. El sistema de ficheros */proc* de Linux también proporciona un subdirectorio por cada proceso en ejecución, cuyo nombre es el ID del proceso. Este subdirectorio contiene ficheros con informaciones tales como la línea de órdenes, enlaces al directorio de trabajo actual y al fichero ejecutable, descriptores de ficheros abiertos, etc. El núcleo proporciona toda esta información al vuelo en respuesta a las peticiones de *read*. Esta implementación no es muy diferente del sistema de ficheros */proc* disponible en Plan 9, pero tiene sus inconvenientes—por ejemplo, para que una herramienta como *ps* liste una tabla de información con todos los procesos en ejecución se debe recorrer un montón de directorios y abrir y leer un montón de ficheros. En comparación, las rutinas *kvm* usadas por otros sistemas UNIX leen directamente las estructuras de datos del núcleo con sólo unas pocas llamadas del sistema.

Obviamente, las diferencias de cada implementación son tan abismales que el convertir las aplicaciones que las usen puede ser una tarea de titanes. Debería resaltarse el hecho de que el sistema de ficheros */proc* de SVR4 es una bestia completamente diferente del sistema de ficheros */proc* que está disponible en Linux y no pueden ser usados en el mismo contexto. No obstante, se puede afirmar que cualquier programa que use las rutinas *kvm* o el sistema de ficheros */proc* de SVR4 no es realmente portable y por tanto dichas secciones de código deben ser reescritas para cada sistema operativo.

La llamada del sistema *ptrace* es casi idéntica a la de BSD, pero hay unas pocas diferencias:

- Las peticiones `PTRACE_PEEKUSER` y `PTRACE_POKEUSER` de BSD se denominan `PTRACE_PEEKUSR` y `PTRACE_POKEUSR`, respectivamente, en Linux.

- Se puede asignar valores a los registros usando la petición `PTRACE_POKEUSR` con los desplazamientos indicados en `/usr/include/linux/ptrace.h`.

- Las peticiones de SunOS `PTRACE_{READ,WRITE}{TEXT,DATA}` no están soportadas, como tampoco lo están `PTRACE_SETACBKPT`, `PTRACE_SETWRBKPT`, `PTRACE_CLRBKPT` o `PTRACE_DUMPCORE`. Las peticiones que faltan sólo deberían afectar a un pequeño número de programas existentes.

Linux *no* proporciona las rutinas *kvm* para la lectura del espacio de direcciones del núcleo desde un programa de usuario, pero algunos programas (notablemente *kmem_ps*) implementan sus propias versiones de estas rutinas.

En general, éstas no son portables, y cualquier código que use las rutinas *kvm* probablemente depende de la disponibilidad de ciertos símbolos o estructuras de datos del núcleo—una suposición poco segura. El uso de las rutinas *kvm* debería considerarse específico de la arquitectura.

10.5 Compilación Condicional Portable

Si necesita hacer modificaciones al código existente para convertirlo a Linux, puede que necesite usar pares `ifdef`... `endif` para rodear las partes del código específicas de Linux, o en general, del código que correspoda a otras implementaciones. No existe un estándar real para seleccionar partes de código a ser compiladas en función del sistema operativo, pero muchos programas usan la convención de definir `SVR4` para el código System V, `BSD` para el código BSD y `linux` para el código específico de Linux.

La biblioteca GNU C usada por Linux le permite activar varias características de la misma definiendo ciertas macros en tiempo de compilación. Estas son:

- `__STRICT_ANSI__`: Sólo características ANSI C.
- `_POSIX_SOURCE`: Características POSIX.1.
- `_POSIX_C_SOURCE`: Si definido a 1, características POSIX.1. Si definido a 2, características POSIX.2.
- `_BSD_SOURCE`: Características ANSI, POSIX y BSD.
- `_SVID_SOURCE`: Características ANSI, POSIX y System V.
- `_GNU_SOURCE`: ANSI, POSIX, BSD, SVID y extensiones GNU. Este es el valor por defecto si no se define ninguna de las anteriores.

Si usted define `_BSD_SOURCE`, se definirá la definición adicional `_FAVOR_BSD` para la biblioteca. Esto hará que ciertas cosas elijan el comportamiento BSD en lugar del comportamiento POSIX o SVR4. Por ejemplo, si está definido `_FAVOR_BSD`, *setjmp* y *longjmp* guardarán y restaurarán la máscara de señales, y *getpgrp* aceptará un parámetro PID. Note que a pesar de todo, sigue teniendo que enlazar con `libbsd` para obtener el comportamiento BSD en las características mencionadas anteriormente en este artículo.

En Linux, `gcc` define un cierto número de macros automáticamente que usted puede utilizar en su programa. Estas son:

- `__GNUC__` (versión GNU C principal, p.ej., 2)
- `__GNUC_MINOR__` (versión GNU C secundaria, p.ej., 5)
- `unix`
- `i386`

- `linux`
- `__unix__`
- `__i386__`
- `__linux__`
- `__unix`
- `__i386`
- `__linux`

Muchos programas usan:

```
#ifdef linux
```

para rodear el código específico de Linux. Usando estas macros de tiempo de compilación puede adaptar fácilmente el código existente para incluir o excluir cambios necesarios para portar el programa a Linux. Nótese que, puesto que Linux incorpora en general más características estilo System V, el mejor código base para comenzar con un programa escrito para System V y BSD es probablemente la versión System V. De forma alternativa, se puede partir de la base BSD y enlazar con `libbsd`.

10.6 Comentarios Adicionales

[1]

Este capítulo cubre la mayoría de los asuntos relativos a la conversión, excepto las llamadas del sistema que faltan y que se indican en el capítulo de llamadas del sistema, así como los *streams* que aún no existen (hay rumores de que debería existir un módulo cargable de *streams* en ftp.uni-stutgart.de en /pub/systems/linux/isdn).

[1] Añadidos por Sven Goldt

Capítulo 11

Llamadas al sistema en orden alfabético

Sven Goldt Guía Linux de Programación

_exit	-	como exit pero con menos acciones (m+c)
accept	-	aceptar conexiones en un socket (m+c!)
access	-	comprobar permisos de usuario en un fichero (m+c)
acct	-	no implementada aun (mc)
adjtimex	-	obtener/ajustar variables de tiempo internas (-c)
afs_syscall	-	reservada para el sist. de ficheros Andrew (-)
alarm	-	envio de SIGALRM tras un tiempo especificado (m+c)
bdflush	-	vuelca buffers modificados al disco (-c)
bind	-	nombrar un socket para comunicaciones (m!c)
break	-	no implementada aun (-)
brk	-	cambiar el tamano del segmento de datos (mc)
chdir	-	cambiar el directorio de trabajo (m+c)
chmod	-	cambiar permisos en un fichero (m+c)
chown	-	cambiar propietario de un fichero (m+c)
chroot	-	cambiar el directorio raiz (mc)
clone	-	vease fork (m-)
close	-	cerrar un fichero (m+c)
connect	-	enlazar dos sockets (m!c)
creat	-	crear un fichero (m+c)
create_module	-	reservar espacio para un modulo del nucleo (-)
delete_module	-	descargar modulo del nucleo (-)
dup	-	crear un duplicado de un descriptor de fichero (m+c)
dup2	-	duplicar un descriptor (m+c)
execl, execlp, execle, ...	-	vease execve (m+!c)
execve	-	ejecutar un fichero (m+c)
exit	-	terminar un programa (m+c)
fchdir	-	cambiar directorio de trabajo por referencia ()
fchmod	-	vease chmod (mc)
fchown	-	cambiar propietario de un fichero (mc)
fclose	-	cerrar un fichero por referencia (m+!c)

fcntl	-	control de ficheros/descriptores (m+c)
flock	-	cambiar bloqueo de fichero (m!c)
fork	-	crear proceso hijo (m+c)
fpathconf	-	obtener info. de fichero por referencia (m+!c)
fread	-	leer matriz de datos de un fichero (m+!c)
fstat	-	obtener estado del fichero (m+c)
fstatfs	-	obtener estado del sistema de ficheros por referencia (mc)
fsync	-	escribir bloques modificados del fichero a disco (mc)
ftime	-	obtener fecha del fichero, en segundos desde 1970 (m!c)
ftruncate	-	cambiar tamano del fichero (mc)
fwrite	-	escribir matriz de datos binarios a un fichero (m+!c)
get_kernel_syms	-	obtener tabla de simbolos del kernel o su tamano (-)
getdomainname	-	obtener nombre de dominio del sistema (m!c)
getdtablesize	-	obtener tamano de la tabla de descriptores de fich. (m!c)
getegid	-	obtener id. de grupo efectivo (m+c)
geteuid	-	obtener id. de usuario efectivo (m+c)
getgid	-	obtener id. de grupo real (m+c)
getgroups	-	obtener grupos adicionales (m+c)
gethostid	-	obtener identificador del huesped (m!c)
gethostname	-	obtener nombre del huesped (m!c)
getitimer	-	obtener valor de temporizador (mc)
getpagesize	-	obtener tamano de pagina (m-!c)
getpeername	-	obtener direccion remota de un socket (m!c)
getpgid	-	obtener id. del grupo de procesos padre (+c)
getpgrp	-	obtener id. del grupo padre del proceso (m+c)
getpid	-	obtener id. del proceso (pid) (m+c)
getppid	-	obtener id. del proceso padre (m+c)
getpriority	-	obtener prioridades de usuario/grupo/proceso (mc)
getrlimit	-	obtener limites de recursos (mc)
getrusage	-	obtener uso de recursos (m)
getsockname	-	obtener direccion de un socket (m!c)
getsockopt	-	obtener opciones ajustadas en un socket (m!c)
gettimeofday	-	obtener segundos pasados desde 1970 (mc)
getuid	-	obtener id. de usuario real (uid) (m+c)
gtty	-	no implementada aun ()
idle	-	hacer candidato a expulsion al disco a un proceso (mc)
init_module	-	incluir un modulo cargable (-)
ioctl	-	manipulacion de un dispositivo de caracter (mc)
ioperm	-	ajusta algunos permisos de e/s (m-c)
iopl	-	ajusta permisos de e/s (m-c)
ipc	-	comunicacion entre procesos (-c)
kill	-	enviar una senal a un proceso (m+c)
killpg	-	enviar una senal a un grupo de procesos (mc!)
klog	-	vease syslog (-!)
link	-	crear un enlace fisico a un fichero (m+c)
listen	-	escuchar conexiones en un socket (m!c)

llseek	-	lseek para ficheros grandes (-)
lock	-	no implementada aun ()
lseek	-	cambia el puntero de un fichero abierto (m+c)
lstat	-	obtiene estado de un fichero (mc)
mkdir	-	crea un directorio(m+c)
mknod	-	crea un dispositivo (mc)
mmap	-	mapea un fichero en memoria (mc)
modify_ldt	-	lee o escribe tabla de descriptores locales (-)
mount	-	montar un sistema de ficheros (mc)
mprotect	-	controla permisos de acceso a una zona de memoria (-)
mpx	-	no implementada aun ()
msgctl	-	control de mensajes ipc (m!c)
msgget	-	obtiene un id. de cola de mensajes (m!c)
msgrcv	-	recibe un mensaje ipc (m!c)
msgsnd	-	envia un mensaje ipc (m!c)
munmap	-	desmapea un fichero de memoria (mc)
nice	-	cambia prioridad del proceso (mc)
oldfstat	-	a extinguir
oldlstat	-	a extinguir
oldolduname	-	a extinguir
oldstat	-	a extinguir
olduname	-	a extinguir
open	-	abrir un fichero (m+c)
pathconf	-	obtener info. de un fichero (m+!c)
pause	-	dormir hasta la llegada de una senal (m+c)
personality	-	cambiar dominio de ejecucion en iBCS (-)
phys	-	no implementada aun (m)
pipe	-	crea una tuberia (m+c)
prof	-	no implementada aun ()
profil	-	perfil de ejecucion (m!c)
ptrace	-	traza proceso hijo (mc)
quotactl	-	no implementada aun ()
read	-	lee datos de un fichero (m+c)
readv	-	lee bloques de un fichero (m!c)
readdir	-	lee un directorio (m+c)
readlink	-	obtener contenido de un enlace simbolico (mc)
reboot	-	reiniciar o controlar combinacion CTRL-ALT-DEL (-mc)
recv	-	recibir mensaje de socket conectado (m!c)
recvfrom	-	recibir mensaje de socket (m!c)
rename	-	mover/renombrar fichero (m+c)
rmdir	-	borrar directorio vacio (m+c)
sbrk	-	vease brk (mc!)
select	-	dormir hasta actividad en un descriptor de fichero (mc)
semctl	-	control de semaforos ipc (m!c)
semget	-	obtener id. de semaforo ipc (m!c)
semop	-	operaciones en conj. de semaforos ipc (m!c)

send	-	enviar mensaje a un socket conectado (m!c)
sendto	-	enviar mensaje a un socket (m!c)
setdomainname	-	ajustar dominio del sistema (mc)
setfsgid	-	ajustar id. grupo del sistema de ficheros ()
setfsuid	-	ajustar id. usuario del sistema de ficheros ()
setgid	-	ajustar id. real de grupo (gid) (m+c)
setgroups	-	ajustar grupos adicionales (mc)
sethostid	-	ajustar identificador de huesped (mc)
sethostname	-	ajustar nombre de huesped (mc)
setitimer	-	ajustar temporizador (mc)
setpgid	-	ajustar id. de grupo padre (m+c)
setpgrp	-	sin efecto (mc!)
setpriority	-	ajustar prioridad de proceso/usuario/grupo (mc)
setregid	-	ajustar id. de grupo real/efectivo (mc)
setreuid	-	ajustar id. de usuario real/efectivo (mc)
setrlimit	-	ajustar limites para los recursos (mc)
setsid	-	crear sesion (+c)
setsockopt	-	cambiar opciones del socket (mc)
settimeofday	-	poner la hora en segundos desde 1970 (mc)
setuid	-	ajustar id. de usuario real (m+c)
setup	-	iniciar dispositivos y montar la raiz (-)
sgetmask	-	vease siggetmask (m)
shmat	-	enganchar memoria a un segm. de memoria compartida (m!c)
shmctl	-	manipulacion de mem. compartida ipc (m!c)
shmdt	-	liberar memoria compartida en un segmento (m!c)
shmget	-	obtener/crear segmento de memoria compartida (m!c)
shutdown	-	desconectar socket (m!c)
sigaction	-	obtener/ajustar manejador de senales (m+c)
sigblock	-	bloquear senales (m!c)
siggetmask	-	obtener senales bloqueadas (!c)
signal	-	poner manejador de senal (mc)
sigpause	-	usar nueva mascara de senales hasta la proxima senal (mc)
sigpending	-	obtener senales bloqueadas pendientes (m+c)
sigprocmask	-	obtener/ajustar mascara de bloqueos de senales (+c)
sigreturn	-	no usada aun ()
sigsetmask	-	ajustar mascara de bloqueos de senales (c!)
sigsuspend	-	reemplaza a sigpause (m+c)
sigvec	-	vease sigaction (m!)
socket	-	crea un extremo de comunicacion para socket (m!c)
socketcall	-	llamada general de sockets (-)
socketpair	-	crea dos sockets conectados (m!c)
ssetmask	-	vease sigsetmask (m)
stat	-	obtener estado del fichero (m+c)
statfs	-	obtener estado del sistema de ficheros (mc)
stime	-	obtener segundos desde 1.1.1970 (mc)
stty	-	no implementada aun ()

swapoff	-	detener el intercambio con un dispositivo o fichero (m-c)
swapon	-	iniciar el intercambio con un dispositivo o fichero (m-c)
symlink	-	crear un enlace simbolico (m+c)
sync	-	volcar bloques modificados a disco (mc)
syscall	-	ejecutar llamada al sistema (-!c)
sysconf	-	obtener valor de una variable del sistema (m+!c)
sysfs	-	obtener info. sobre sistemas de ficheros usados ()
sysinfo	-	obtener info. sobre el sistema (m-)
syslog	-	manipulacion del registro (m-c)
system	-	ejecutar un comando de shell (m!c)
time	-	obtener segundos desde 1.1.1970 (m+c)
times	-	obtener tiempos del proceso (m+c)
truncate	-	cambiar tamano de un fichero (mc)
ulimit	-	obtener/ajustar limites de fichero (c!)
umask	-	ajustar mascara de creacion de ficheros (m+c)
umount	-	desmontar un sistema de ficheros (mc)
uname	-	obtener info. del sistema (m+c)
unlink	-	borrar un fichero no bloqueado (m+c)
uselib	-	usar libreria compartida (m-c)
ustat	-	no implementada anu (c)
utime	-	motificar info. de tiempo en nodo-i (m+c)
utimes	-	vease utime (m!c)
vfork	-	vease fork (m!c)
vhangup	-	colgar virtualmente el terminal actual (m-c)
vm86	-	entrar en modo vm86 (m-c)
wait	-	esperar terminacion de proceso (m+!c)
wait3	-	espera terminacion de un proceso (bsd) (m!c)
wait4	-	espera terminacion de un proceso (bsd) (mc)
waitpid	-	espera terminacion de un proceso (m+c)
write	-	escribir datos a un fichero (m+c)
writev	-	escribir bloques de datos a un fichero (m!c)

(m) hay página de manual.
(+) cumple norma POSIX.
(-) Específica de Linux.
(c) de libc.
(!) no es solo una llamada al sistema. Usa otras

Sven Goldt The Linux Programmer's Guide

Capítulo 12

Abreviaturas

ANSI	American National Standard for Information Systems *(Estándar Nacional Americano para Sistemas de Información)*
API	Application Programming Interface *(Interfaz de Programación de Aplicaciones)*
ASCII	American Standard Code for Information Interchange *(Código Estándar Americano para el Intercambio de Información)*
AT 386	Advanced Technology Intel 80386 (PC basado en 386, "*tecnología avanzada*")
FIPS	Federal Information Processing Standard *(Estándar Federal de Proceso de Información)*
FSF	Free Software Foundation *(Fundación para el Software Libre)*
IEEE	Institute of Electrical and Electronics Engineers, Inc. *(Instituto de Ingenieros de Electricidad y Electrónica)*
IPC	Inter Process Communication *(Comunicación entre Procesos)*
ISO	International Organization for Standards *(Organización Internacional de Estándares)*
POSIX	Portable Operating System Interface for uniX *(Interfaz de programación transportable entre sistemas operativos UniX)*
POSIX.1	IEEE Std. 1003.1-1990 Estándar de Tecnología de Información - Interfaz para Portabilidad entre Sistemas Operativos (POSIX) - Part 1: Interfaz de Programación de Aplicaciones (API)