UNIX

Entorno de Programación

Un entorno para programadores
Expresiones regulares
Lenguajes patrón-acción
La *shell* un lenguaje de programación
Características de la prog. *shell* Korn
Introducción al lenguaje C
Compilación de C
Compilación de proyectos
Herramientas de desarrollo
Depuración de programas
Control de versiones
Tratamiento de procesos
Señalización entre procesos

INDICE

CAPÍTULO 1

Un entorno para programadores

CAPÍTULO 2

Metacaracteres y expresiones regulares

CAPÍTULO 3

Lenguajes patrón-acción

INDICE

INDICE

INDICE

INDICE

CAPÍTULO 7

Compilación de C

CAPÍTULO 8

Compilación de proyectos

INDICE

CAPÍTULO 9

Herramienta de desarrollo (cscope)

CAPÍTULO 10

Depuración de programas

CAPÍTULO 11

Control de versiones

INDICE

CAPÍTULO 12

Tratamiento de procesos

CAPÍTULO 13

Señalización entre procesos

INDICE

APÉNDICE A

Programas para prácticas

APÉNDICE B

Soluciones

CAPÍTULO 1

UN ENTORNO PARA PROGRAMADORES

1.1 Objetivos del curso

Unix es un sistema operativo muy apropiado para el desarrollo de aplicaciones. El entorno flexible que proporciona más el conjunto de herramientas y utilidades disponibles conforman uno de los sistemas más utilizados en el mundo en la tarea de especificación, construcción, verificación y pruebas de muy diversos tipos de aplicaciones.

Este curso se propone presentar todos aquellos aspectos útiles para abordar la creación de una nueva aplicación. No pretende lógicamente exponer el nivel último de detalle (puesto que ese es el objetivo de los manuales de referencia), sino dar a conocer cuáles son las opciones disponibles y la ciencia necesaria para saber enfocar y emprender, con muchas probabilidades de éxito, la realización de tanto pequeñas herramientas como complejas aplicaciones.

1.2 Soluciones Unix para el desarrollo de aplicaciones

Existen varias alternativas para la creación de un programa de tamaño pequeño, medio o grande bajo el sistema Unix.

A. LENGUAJE C

Una posibilidad es la de construir la aplicación directamente en el lenguaje de programación C. La mayor parte del sistema Unix está escrita en C, por lo que existe un gran entendimiento entre los dos. En Unix están disponibles un número importante de herramientas muy útiles para la edición, compilación y depuración de programas en C.

En este curso analizaremos algunas de las más utilizadas, como por ejemplo, el compilador `cc`, la utilidad `make` para compilar proyectos de forma automática, la utilidad de depuración `sdb`, etc. Todas estas utilidades suelen incorporarse por defecto en el propio sistema operativo.

Antes de decidir utilizar C para construir una aplicación se deberían considerar una serie de cuestiones:

- El tiempo que llevaría la realización del programa directamente en C.
- Si no existe alguna utilidad ya disponible en el sistema que realice la acción que queremos.
- Si no sería más rápido la construcción de uno o varios programas de shell en vez de programar en C.
- Si aunque lo anterior fuera posible, se necesita la rapidez de ejecución que se puede obtener con un programa C optimizado.

B. PROGRAMACIÓN EN SHELL

La shell con la que los usuarios interactúan proporciona la posibilidad de construir programas completos mediante mandatos y estructuras de programación de la propia shell.

Al principio, un usuario no experimentado en programación realiza sus primeros ensayos listando en un fichero unos cuantos mandatos. De esta forma ahorra tener que ejecutarlos uno tras otro. Pronto descubrirá que puede confeccionar de esta manera grandes y complejas tareas.

La shell es una herramienta muy potente para la realización de aplicaciones, y tiene la ventaja de disponer de un gran número de mandatos y utilidades que pueden ser invocados directamente desde un fichero de texto. Además, como en todo lenguaje de programación, existen sentencias estructuradas que le proporcionan una potencia considerable.

Todas estas facilidades dan lugar a otra de las ventajas de la programación en shell: la relativa rapidez con la que es posible desarrollar herramientas y aplicaciones. Esta ventaja está posiblemente contrarrestada con un inconveniente: un programa en shell puede que no se ejecute con la rapidez requerida.

En este curso presentaremos las estructuras sintácticas y otros muchos aspectos de la programación en shell. Aunque nos centraremos en la shell de Korn por ser la que tiene mejores prestaciones actualmente, indicaremos diferencias existentes con las restantes.

C. UTILIDADES Y HERRAMIENTAS

Programar en shell posibilita la utilización de herramientas disponibles en la misma. Estas utilidades están orientadas a la realización de acciones muy particulares: buscar patrones en un flujo de caracteres, buscar patrones y realizar acciones, transformar un flujo de entrada en uno de salida diferente, extraer información ubicada en un lugar concreto de un fichero, etc.

La combinación de varias de estas utilidades entre sí mediante cauces ("pipes"), y el uso adecuado de los canales estándares (entrada, salida y error), constituyen una opción a tener en cuenta a la hora de construir una aplicación.

En este curso presentaremos varias utilidades y herramientas simples (algunas de ellas no tan simples, con la entidad incluso de un lenguaje de programación).

D. APLICACIONES MULTIPROCESO

En algunas situaciones es posible que sea muy conveniente o incluso necesario dividir la aplicación en varias tareas independientes, y utilizar algún mecanismo de comunicación para pasar información entre las mismas cuando sea necesario.

El sistema operativo Unix dispone de:

- Posibilidad de activar tareas (procesos) que se ejecuten independientemente.
- Facilidad de controlarlos y manejarlos adecuadamente.
- Múltiples mecanismos de comunicación entre procesos.

Además de diversos mandatos para controlar procesos y trabajos desde la shell, Unix proporciona un número importante de servicios, normalmente conocidos como llamadas al sistema, que permiten un control exhaustivo sobre los mismos. Estas llamadas al sistema deben ser invocadas desde programas en C, y desde el punto de vista del programador son utilizadas como simples funciones C.

En este curso explicaremos algunas de las llamadas al sistema esenciales para el manejo de procesos y para la intercomunicación entre ellos.

E. UNA SOLUCIÓN MIXTA

En la mayoría de las situaciones se suelen hacer combinaciones de las alternativas posibles. En muchos casos, alguna tarea puede implementarse fácilmente mediante un programa de shell o directamente ser realizada por una herramienta disponible en el sistema. En otros casos es necesario descender a tan bajo nivel que se requiere trabajar en C.

Desde un programa en C se puede activar otro programa, tanto ejecutable como un programa de shell, y lógicamente lo contrario también es posible, activar (tanto en primer plano como en segundo plano) programas ejecutables que han sido desarrollados en C.

Todas estas facilidades e interrelaciones favorecen una solución mixta, por ejemplo la elaboración de aplicaciones desarrolladas en C que son activadas mediante "scripts" de shell.

1.3 Un entorno de programación

Una afirmación que se suele hacer con bastante frecuencia es el atractivo que Unix tiene especialmente para programadores. Es decir, aquellas personas que están acostumbradas a programar se sienten cómodas en este entorno, aprenden con rapidez y utilizan los mandatos y sus posibilidades de combinanción con cierta soltura.

La justificación de este hecho está en parte fundada en que el sistema operativo Unix fue creado por programadores para programadores. Ken Thompson y Dennis Ritchie recibieron el reconocimiento a su trabajo en la creación de este sistema con la concesión del premio Turing en 1983. Algunos de los comentarios con motivo de este premio fueron los siguientes:

> *"... El modelo del sistema Unix ha conducido una generación de diseñadores software a nuevas formas de pensar sobre programación. La genialidad del sistema Unix es su estructura, la cual permite a los programadores avanzar a partir del trabajo de otros."*

El dato más relevante para entender el porqué de su atractivo para los programadores es tener en cuenta que Unix fue pensado como un entorno de programación cómodo para sus propios creadores, sin pensar por el contrario en un sistema con pretensiones comerciales.

En este curso intentaremos demostrar todas estas afirmaciones, proporcionando al asistente la suficiente información y experiencia como para que juzgue por sí mismo.

CAPÍTULO 2

METACARACTERES Y EXPRESIONES REGULARES

1.1 Conceptos de metacarácter y patrón de caracteres

Muchas veces surge la necesidad de referirse de forma precisa y compacta a toda una familia (posiblemente infinita) de cadenas de caracteres.

Por ejemplo, el conjunto de cadenas que empiezan por `b` o `B`.

Una estrategia ampliamente utilizada consiste en dotar de significado especial a ciertos caracteres (metacaracteres) de forma que una cadena sin metacaracteres representa la familia compuesta por ella misma, mientras que una cadena con metacaracteres representará una familia más grande en algún sentido.

Un ejemplo aclarará este concepto. Para la shell, una orden como

```
rm  B*
```

borrará todos los ficheros del directorio en curso cuyo nombre empiece por `B`, y no un fichero llamado "`B*`". Se dice entonces que `*` es un metacarácter cuyo significado es "*cualquier cadena, etc.*".

Esta notación, por desgracia, no es general en todos los contextos, aunque sí podríamos decir que existen muchas similitudes. En el caso del sistema operativo Unix debemos destacar dos mundos distintos pero parecidos: el de los metacaracteres de la shell, y el de las expresiones regulares.

Para no confundirse "demasiado", separaremos estos dos mundos en este capítulo y a lo largo de todo el curso, con el objetivo de conocer bien la distinción de un metacarácter en uno y en otro lugar.

Normalmente los nombres de ficheros se referencian mediante metacaracteres de la shell, y las utilidades que realizan una búsqueda en texto suelen usar expresiones regulares. Por ejemplo, los mandatos `find` y `ls` utilizan metacaracteres de la shell para especificar nombres de ficheros. Mientras que por ejemplo el editor `vi`, las herramientas `sed` y `awk` usan expresiones regulares.

1.2 Metacaracteres de la shell. Nombres de ficheros

La shell utiliza caracteres especiales para construir patrones que especifiquen nombres de ficheros, y además caracteres especiales para especificar acciones y modos de funcionamiento.

A continuación se listan todos los caracteres con un significado especial:

```
*     ?       []      [!]
>     2>      >>      2>>     <       <<      >&      |
' '           " "             ` `
()            &               $       \
(( ))         [[ ]]           &&      ||
```

A. NOMBRES DE FICHEROS

De entre todos estos metacaracteres existen algunos que se utilizan para describir nombres de ficheros, recordamos a continuación su uso mediante algunos ejemplos:

*	Cualquier secuencia de caracteres.
?	Cualquier carácter (sólo uno).
[...]	Una clase de caracteres.
[!]	Complementario de una clase de caracteres.

Prácticas

```
$ ls -l c*
$ ls -l ?????
$ ls -ld [mps]*
$ ls -ld [!mps]*
$ ls -ld *[ae]*
$ ls -ld '.*
```

B. ESCAPAR METACARACTERES

La shell analiza el mandato que el usuario teclea antes de ejecutarlo. En este análisis la shell interpreta todos los metacaracteres que aparezcan en la línea de mandatos. Por ejemplo, en el caso de listar ficheros mediante el mandato `ls c*`, la shell buscará en el directorio actual todos aquellos ficheros que comiencen por c, situará los nombres de estos ficheros en la línea de mandatos, y finalmente lo ejecutará.

En muchas ocasiones es necesario que la shell NO interprete alguna parte de la línea de mandatos, ya que la sustitución no procede. Por ejemplo pensemos en el nombre de un fichero en el que aparece algún metacarácter de la shell. En este caso, interesa que la shell no interprete el nombre de ese fichero. Para escapar metacaracteres existen tres posibilidades:

- CUALQUIER CARÁCTER que vaya precidido por un carácter "\" no será interpretado por la shell. En particular si ese carácter es un metacarácter, perderá su significado especial.
- Una cadena encerrada entre comillas simples ´ ´, nunca será interpretada por la shell, con lo cual dicha cadena se pasará sin ninguna modificación al mandato correspondiente.
- Una cadena encerrada entre comillas dobles " " será interpretada parcialmente por la shell. En concreto los únicos metacaracteres que interpreta dentro de comillas dobles son "\" (escapar sólo metacaracteres), "$" (acceso a variables de shell) y ` ` (comillas de sustitución de mandatos). Cualquier otro carácter no será interpretado. En concreto el carácter "\" se utiliza para escapar únicamente a los tres caracteres que tienen un significado especial entre comillas dobles. En cualquier otro caso, el carácter "\" no tendrá ningún significado especial.

Prácticas

1. Cree un fichero cuyo nombre contenga algún metacarácter. Por ejemplo:

```
$ cat > \*
.....
^D
```

Intente ahora visualizarlo (`more` o `cat`), listarlo,... Para borrarlo ejecute:

```
$ rm -i \*
```

2. Ejecute las siguientes órdenes y explique el resultado:

```
echo \n
echo '\n'
echo "\n"
echo '$HOME'
echo "$HOME"
echo "\*"
echo "\$HOME"
```

1.3 Expresiones regulares

Una expresión regular es una notación formal de un conjunto de cadenas de caracteres. El concepto de expresión regular es el mismo que el expuesto para los metacaracteres de la shell: utilizar caracteres especiales para denotar cadenas de caracteres.

Podríamos hablar también de expresiones regulares en el caso de los metacaracteres de la shell, sin embargo no lo hacemos por dos motivos:

(a.) Porque los caracteres especiales que se utilizan en la shell no tienen el mismo significado que en otros contextos, donde sí se habla de expresiones regulares.

(b.) Para ayudar a la comprensión y a la no confusión entre metacaracteres o comodines de la shell, y expresiones regulares usadas en utilidades, e insistir en sus diferencias.

A continuación listamos los caracteres especiales utilizados en expresiones regulares:

Metacarácter	Significado	Ejemplo	Explicación
.	Cualquier carácter salvo el salto de línea	`x.z`	Una x seguida de cualquier carácter y después z.
^	Principio de línea	`^hola`	La cadena `hola`, sólo si está al principio de una línea
$	Final de línea.	`hola$`	La cadena `hola`, sólo si está al final de una línea
*	Cero o más ocurrencias	`a*`	Ninguna, una o varias repeticiones de `a`
+	Una o más ocurrencias	`a+`	una o varias repeticiones de `a`
[]	Cualquier carácter en [ ]	`[xA0]`	Una `x`, una `A` o un `0`
[c-d]	Cualquier carácter en el rango de `c` a `d`, ambos inclusive	`[x-z]`	La `x`, la `y` o la `z`.
[^]	Cualquier carácter que no esté en [^]	`[^0-9]`	Cualquier carácter que no sea dígito.
\	Anula el significado especial del siguiente carácter	`\+`	El carácter +
&	Equivale a la cadena encontrada en una sustitución (sólo en `ed`, `vi` y `sed`)	s/H../&ABC/	Añade `ABC` a la cadena que se encuentre (`H` seguida de cualesquiera dos caracteres).
\{n\}	`n` repeticiones de la expresión anterior	`e\{5\}`	5 e's consecutivas.
\(er\)	Se pueden referenciar las ocurrencias de la expresión regular `er` por \n, donde *n* es el número de orden de expresiones de este tipo	`\(.\).\1`	Palíndromos de tres caracteres.
\<	Principio de palabra	`\<bi`	Palabras que empiezan por `bi`.
\>	Final de palabra	`en\>`	Palabras que terminan en `en`.

Observe como en los 4 últimos casos el carácter "\" en vez de anular el posible significado especial del carácter que le sigue, funciona a la inversa, es decir, proporcionándole un tratamiento especial.

1.4 Prácticas con expresiones regulares

A. PRÁCTICAS CON GREP.

Practique con los siguientes ejemplos de expresiones regulares con `grep`, y explique sus resultados:

```
$ grep "^[a-z]r" fichgrep
$ grep ".*" fichgrep
$ grep '^\(.\).*\1$' fichgrep
$ grep -v "[g-z]" fichgrep
$ grep "[^g-z]" fichgrep
$ grep "\<el\>" fichgrep
$ grep "\A" fichgrep
$ grep "\+" fichgrep
```

B. PRÁCTICAS CON EGREP

El mandato `egrep` permite algunos caracteres especiales más:

Metacarácter	Significado	Ejemplo	Explicación
+	Una o más ocurrencias	`a+`	una o más `a's`
?	Cero o una ocurrencia	`a?`	La cadena nula o una `a`
( )	Agrupar expresiones regulares	`([0-9]a)*`	Repetición cero o más veces de una secuencia de un carácter seguido de `a`.
|	Opción de expresiones regulares	`ab|^c`	`ab` o una línea que empiece por `c`.

```
$ egrep "([0-9]a[rst])+" fichegrep
$ egrep "ab|^[0-9]a*|ab?" fichegrep
```

C. PRÁCTICAS CON VI

A continuación aparecen algunos ejemplos de utilización de expresiones regulares con búsqueda y sustitución mediante vi:

```
/cadena          #Buscar una cadena.
/^Al             #Buscar la cadena "Al" al principio de
                 línea.
/final$          #Buscar la cadena "final" al final de
                 una línea.
/\$              #Buscar el carácter $.
:s/Unix/UNIX     #Sustituir la primera ocurrencia de
                 Unix en la línea actual, por UNIX.
:s/Unix/UNIX/g   #Sustituir todas las ocurrencias de
                 Unix en la línea actual por UNIX.
:s/U..[xX]/el sistema/g
:s/^/Principio/
:1,9s/^/Principio/
:1,$s/^/Principio/
:$s/$/Final/
:1,$s/a.a/&AQUI/g
:1,$s/\([0-9]\)[0-9]\1/&CAPI/g
```

CAPÍTULO 3

LENGUAJES PATRÓN-ACCIÓN

1.1 Procesamiento de texto

El sistema Unix dispone de varias herramientas para procesar el contenido de un fichero. A muchas de estas herramientas se les suele conocer con el nombre de filtros, ya que leen su entrada estándar, la procesan, y escriben en la salida estándar.

Estas herramientas se pueden utilizar para manejar pequeñas bases de datos, organizando la información mediante simples ficheros planos. Constituyen una forma rápida de construir una pequeña aplicación de base de datos. Estas herramientas por tanto proporcionan una gran potencia al sistema Unix. Evidentemente nunca pueden compararse con los grandes Sistemas de Gestión de Bases de Datos actuales, ya que el acceso y las operaciones de búsqueda son siempre secuenciales con el consiguiente retardo.

Existen muchas órdenes en Unix para procesar ficheros que supondremos conocidas. A continuación enumeramos algunas de ellas:

more, cat, pg	visualizar la salida de un fichero.
head, tail	visualizar la cabeza y la cola de un fichero.
od, hd	visualizar el contenido de un fichero en formato octal y hexadecimal, respectivamente.
diff, cmp, comm	comparar ficheros.
cut	extraer verticalmente (cortar) campos de un fichero.
sort	ordenar líneas de un fichero.
uniq	obtener las líneas "únicas" (no repite líneas iguales) de un fichero ya ordenado.

1.2 Lenguajes patrón-acción

Dos de los filtros más importantes son `sed` y `awk`, de los que hablaremos en este capítulo. Estas dos herramientas se clasifican en lo que se suelen llamar lenguajes patrón-acción. Esta definición indica que la utilidad buscará por los patrones (descritos habitualmente mediante expresiones regulares) que se le indiquen, y una vez encontrados ejecutará la acción especificada.

Esta filosofía no sólo aparece en Unix sino que es un área de trabajo clásica en informática. La descripción de los patrones y de las acciones que se quieren realizar puede ampliarse y potenciarse tanto, que en algunos casos incluso se habla de lenguajes de programación. Un ejemplo claro es la utilidad `awk`, que permite utilizar sentencias de programación estructurada en las acciones.

Otra herramienta que también entraría en esta definición es la utilidad `lex`. Aunque está pensada para ser combinada con otra utilidad disponible en Unix (`yacc`), de forma que sea relativamente cómoda la construcción de analizadores sintácticos, es también una herramienta de uso independiente, en donde las acciones son descritas mediante el lenguaje de programación C. Aunque son herramientas de desarrollo, los fundamentos teóricos necesarios para abordar con comodidad su comprensión están fuera del alcance de este curso, por lo que hemos decidido no introducirlas en el mismo.

1.3 El editor de flujo sed

La herramienta sed es un editor de flujo derivado a partir del editor ed. Funciona como un filtro que lee una a una las líneas de los ficheros de entrada, y las transforma aplicándoles las acciones que se hayan especificado. Hay que tener en cuenta que sed nunca modifica el fichero original, la transformación se envía a la salida estándar. Si ésta se quiere recoger, puede hacerse redireccionándola a un fichero, o utilizando la orden w de sed.

1.4 La orden sed

La sintaxis del mandato `sed` es la siguiente:

`sed [-n] [-e mandato] [-f fich_man ] [fich1 [fich2] ... ]`

donde,

`-n`	Por defecto, el mandato `sed` envía a la salida estándar todas las líneas después de procesarlas. La opción `-n` desactiva este funcionamiento, de forma que sólo se enviarán a la salida estándar aquellas líneas que se especifiquen explícitamente en la parte de acciones.
`-e mandato`	Con esta opción se indica a `sed` qué *mandato* (patrón-acción) se quiere ejecutar. No es necesario precederlo con `-e` si no aparece la opción `-f`. El *mandato* se suele encerrar entre comillas simples puesto que en él pueden aparecer caracteres con significado especial para la shell.
`-f fich_man`	Mediante la opción `-f` se puede especificar un fichero de mandatos para `sed`.
fichX	`sed` leerá una a una las líneas de cada uno de los ficheros que se le pasen como argumento, o de su entrada estándar. Sobre cada una de esas líneas aplicará secuencialmente todos los mandatos que se hayan especificado, bien mediante la opción `-e`, y/o en un fichero de mandatos.

1.5 Mandatos sed

El formato de los mandatos de `sed` es el siguiente:

`[línea1 [, línea2]] acción [argumentos]`

donde

`líneaX` Especifica una línea donde se va a ejecutar la acción que se describe a continuación. Esta línea se puede especificar de las siguientes formas:

- Un número decimal que indica el número de línea en el texto de entrada.
- El carácter $ indicando la última línea.
- Una expresión regular encerrada entre /. Esto quiere decir que se realizaría la acción sobre cada línea que contuviese dicha expresión regular.

`línea1, línea2` Mediante la separación de dos líneas por una coma se especifica un rango de líneas donde se realizará la acción que se describe a continuación.

`acción [argumentos]` Especifica qué acción se realizará sobre cada línea. Los tipos de acciones son los siguientes:

Mandato	Explicación
p	imprimir líneas
d	borrar líneas
s	sustituir cadenas
q	abandonar el procesamiento
i	insertar líneas
c	reemplazar líneas
a	añadir líneas
y	sustituir caracteres
w fich	escribir líneas en un fichero
r fich	leer un fichero

1.6 Imprimir líneas

El mandato p es utilizado por sed para imprimir líneas por pantalla. Por defecto sed ya imprime cada línea que procesa, por lo tanto las líneas que se impriman mediante la orden p se visualizarán dos veces.

Veamos los siguientes ejemplos:

```
$ cat osilinea
1  El modelo OSI de ISO
2  Puesto que necesita una estructura donde colgar los acronimos
3  y palabras claves, debe conocer primero la ISO y su modelo OSI.
4  La Organizacion Internacional de Estandares (ISO, Inter-
5  national Standards Organization) situada en Paris, desarrolla
6  normativas para comunicacion de datos nacionales e interna-
7  cionales. El representante americano en ISO es el Instituto Na-
8  cional Americano de Estandares (American National Standards
9  Institute, ANSI). Al principio de la decada de los setenta, la ISO
10 desarrollo un modelo estandar de sistema de comuinicacion de datos
11 y lo llamo el modelo OSI, Open Systems Interconnection
12 (sistema de interconexion abierto).
Ultima linea numero 13
$ sed  '' osilinea
```

Esta última orden visualizaría todo el fichero sin ningún cambio, ya que no se realiza ninguna acción sobre ninguna línea del mismo, y por defecto todas las líneas son enviadas a la salida estándar. Sin embargo la siguiente orden no imprimiría nada por la pantalla:

```
$ sed -n '' osilinea
```

Si utilizamos el mandato p, sin poner ninguna condición a la línea sobre la que se aplica, se imprimirían por pantalla todas las líneas del fichero (dos veces si no se utiliza -n, y una si se utiliza).

```
$ sed 'p' osilinea
$ sed -n 'p' osilinea
```

Como hemos comentado anteriormente, a una acción puede acompañarle una referencia a una "línea" del fichero. Donde esa línea podía ser un número, $ o una expresión regular. Veamos algunos ejemplos para el caso del mandato p.

```
$ sed -n '3p' osilinea
```

Imprime la línea número 3 del fichero.

```
$ sed -n '$p' osilinea
```

Imprime la última línea del fichero.

```
$ sed -n '/OSI/p' osilinea
```

Imprime aquellas líneas que contienen la cadena "OSI". Las expresiones regulares deben encerrarse mediante "/".

1.7 Borrar líneas

El mandato d borra las líneas especificadas. Tenga en cuenta que esto significa exclusivamente no imprimir esas líneas en la salida estándar, pero que en ningún caso sed modifica el fichero.

```
$ sed  '7,11d' osilinea
1  El modelo OSI de ISO
2  Puesto que necesita una estructura donde colgar los acronimos
3  y palabras claves, debe conocer primero la ISO y su modelo OSI.
4  La Organizacion Internacional de Estandares (ISO, Inter-
5  national Standards Organization) situada en Paris, desarrolla
6  normativas para comunicacion de datos nacionales e interna-
12 (sistema de interconexion abierto).
Ultima linea numero 13
$ sed '/u/d' osilinea
1  El modelo OSI de ISO
4  La Organizacion Internacional de Estandares (ISO, Inter-
8  cional Americano de Estandares (American National Standards
11 y lo llamo el modelo OSI, Open Systems Interconnection
12 (sistema de interconexion abierto).
```

1.8 Sustituir cadenas

El formato del mandato de sustitución (s) de `sed` es:

s/expresión regular/sustituciones/modificadores

lo cual significa que se sustituirá cualquier ocurrencia de una cadena que cumpla *`expresión regular`* por *`sustituciones`*. Los *`modificadores`* que se admiten son los siguientes:

n	Sustituir sólo la n-ésima ocurrencia.
g	Global. Sustituir todas las ocurrencias de la línea.
p	Imprimir la línea despúes de realizar la sustitución.
w *fich*	Añadir la línea al fichero *fich*, después de realizar la sustitución.

Veamos algunos ejemplos:

```
$ sed 's/e/OTRAE/2' osilinea
$ sed '2,3s/e/OTRAE/g'osilinea
$ sed "/claves/,9s/^[2-8]/&  PRINCIPIO/" osilinea
$ sed -n '/-$/s/^.*$/guion/p'osilinea
$ sed -e "/OSI/,/ISO/s/^/       /" osilinea
```

1.9 Abandonar el procesamiento

Otra posibilidad es la de abandonar el procesamiento de líneas de entrada cuando suceda alguna condición, por ejemplo la ocurrencia de una expresión regular o un número de línea. El mandato `sed` para realizar esto es `q`. Veamos algunos ejemplos:

```
$ sed "8q" osilinea
1  El modelo OSI de ISO
2  Puesto que necesita una estructura donde colgar los acronimos
3  y palabras claves, debe conocer primero la ISO y su modelo OSI.
4  La Organizacion Internacional de Estandares (ISO, Inter-
5  national Standards Organization) situada en Paris, desarrolla
6  normativas para comunicacion de datos nacionales e interna-
7  cionales. El representante americano en ISO es el Instituto Na-
8  cional Americano de Estandares (American National Standards
```

Tenga en cuenta que el procesamiento de líneas parará justo en el momento en el que se aplique una acción `q` sobre una línea, esto quiere decir que cualquier otra acción que le siga, no se ejecutará. Veamos el siguiente ejemplo en el que se utiliza un fichero de mandatos de `sed` para especificar las acciones que se quieren realizar.

```
$ cat fmanq
8q
7,8s/o/OO/g
$ sed -f fmanq osilinea
1  El modelo OSI de ISO
2  Puesto que necesita una estructura donde colgar los acronimos
3  y palabras claves, debe conocer primero la ISO y su modelo OSI.
4  La Organizacion Internacional de Estandares (ISO, Inter-
5  national Standards Organization) situada en Paris, desarrolla
6  normativas para comunicacion de datos nacionales e interna-
7  ciOOnales. El representante americanOO en ISO es el InstitutOO Na-
8  cional Americano de Estandares (American National Standards
```

1.10 Insertar, añadir y reemplazar líneas

La orden sed también dispone de mandatos para insertar una línea antes de otra (i), añadir una línea después de otra (a), y reemplazar una línea por otra (c). Cuando se teclea una orden sed que utiliza cualquiera de estos mandatos, el texto a insertar, añadir o reemplazar debe introducirse después de la secuencia "\<retorno>" (donde <retorno> quiere decir pulsar la tecla retorno de carro). Veamos algunos ejemplos:

```
sed '/OSI/i\<retorno>
En la siguiente línea aparece OSI' osilinea
sed '3i\<retorno>
Texto insertado antes de la línea 3'osilinea
sed '/OSI/a\<retorno>
En la línea anterior aparece OSI'osilinea
sed '3,6a\<retorno>
Texto despues de la 3, 4, 5 y 6' osilinea
sed '/OSI/c\<retorno>
Línea reemplazada' osilinea
sed '4c\<retorno>
Nueva línea 4' osilinea
```

1.11 Sustituir caracteres

El mandato y de `sed` permite realizar una sustitución de un carácter por otro. El formato de y es:

`y/cadena1/cadena2/`

donde el carácter que ocupa la posición i-ésima en `cadena1`, se sustituirá por el carácter que ocupa la misma posición i-ésima en `cadena2`. Esto quiere decir que la longitud de las dos cadenas debe ser la misma. veamos algunos ejemplos:

```
sed -e "y/aeiou/AEIOU/" osilinea
```

Sustituirá las vocales en minúsculas por las vocales en mayúsculas.

```
sed -e "/que/s/a/AHA/g" osilinea
```

Sustituirá todas las "a" por "AHA" en las líneas que contengan "que". Analice los siguientes ejemplos:

```
sed -e "5,$y/abcde/ABCDE/" osilinea
sed -e "/OSI/,12y/aeiou/AEIOU/" osilinea
```

1.12 Escribir y leer en un fichero

Los mandatos `w` y `r` de `sed` permiten escribir y leer en un fichero. El funcionamiento es el siguiente:

w *fich*	Las líneas que cumplan las condiciones que se especifiquen para esta acción serán añadidas al fichero *fich*. La primera vez que se escribe en el fichero se borrará, y las siguientes escrituras serán para añadir al final del mismo.
r *fich*	El contenido del fichero *fich* es añadido a la salida estándar, después de las líneas que cumplan las condiciones que se especifiquen para esta acción.

Veamos algunos ejemplos:

```
$ sed '/OSI/w wosi' osilinea
$ sed '3,5w wosi' osilinea
$ sed '3,6r wosi' osilinea
$ sed '$r wosi' osilinea
```

1.13 Ficheros de mandatos

Como hemos visto anteriormente es posible construir un fichero de mandatos para `sed`. Esto permite construir múltiples acciones sobre las líneas de entrada. Estas acciones se aplican secuencialmente sobre cada línea, es decir en el orden en el que hayan sido escritas en el fichero de mandatos. Supongamos que el fichero denominado `fmansed` contiene las siguiente acciones.

```
s/OSI/osi/g

s/i/I/g
```

Este fichero se puede usar en el mandato:

```
$ sed -f fmansed osilinea
```

Para cada línea del fichero `osilinea`, primero se sustituirán todas las cadenas "OSI" por "osi", y después, cada "i" por "I". Por ello el resultado será "osI".

Las líneas en el fichero de mandatos que empiezan por "**#**" son líneas de comentario y se ignoran. Una excepción es "**#n**", que funciona del mismo modo que la opción `-n` en la línea de mandatos, es decir eliminando la acción por defecto de imprimir por la salida estándar.

1.14 El operador !

Este operador puede aplicarse sobre cualquier acción. El efecto que produce es la ejecución de la acción indicada sobre las líneas que no hayan sido especificadas. Es decir tiene un efecto de negación.

Por ejemplo,

```
$ sed -n '/OSI/!p' osilinea
```

esta orden visualiza las líneas que no contienen la cadena OSI. El siguiente ejemplo mostraría el texto sin cambios:

```
$ sed '!s/a/AAAAA/g' osilinea
```

¿Cuál sería el efecto de la siguiente orden?

```
$ sed '/OSI/!q' osilinea
```

1.15 Prácticas

1. Añada la palabra PRIMERO al principio de la primera línea del fichero `osilinea`.
2. Escriba una orden `sed` para eliminar los espacios en blancos al final de cada línea.
3. Escriba una orden `sed` para eliminar el último carácter del fichero.
4. Escriba una orden `sed` para eliminar la última línea de un fichero.
5. Utilice `sed` para averiguar los mandatos que están en segundo plano en su cuenta.
6. Utilice `sed` para extraer sólo los permisos de todos los ficheros en el directorio actual.

1.16 Lenguaje awk

La herramienta awk es un potente lenguaje de procesamiento, tanto que todo lo que se puede hacer con sed puede hacerse con awk. Sin embargo, este último tiene un procesamiento orientado a campos que le proporciona capacidades adicionales. Las acciones que se describen en awk tienen una sintaxis parecida a C, e incluyen muchas de las sentencias estructuradas de un lenguaje de programación, lo que permite describir instrucciones muy complejas.

Este lenguaje fue creado por Aho, Weinberger y Kernighan en 1977 como un lenguaje de búsqueda de patrones. Desde entonces se han añadido nuevas características a esta utilidad. Una versión importante apareció con la Versión 3.1 de Unix Sistema V. La versión actual se conoce como nawk. El renombramiento de esta utilidad es temporal, con el objetivo de dar tiempo para actualizar las aplicaciones que usaban awk. Para versiones futuras se prevé que no existan ya diferencias, y que la orden awk ejecute la versión nawk actual.

Algunas de las novedades de esta versión nawk son las siguientes:

- caracteres de ocho bits,
- definición de funciones por el usuario,
- más variables predefinidas,
- mejora en la sintaxis de los subíndices de matrices,
- la función system() para ejecutar mandatos en una shell.

En de este capítulo escribiremos awk, y destacaremos cuándo un ejemplo sólo funciona con nawk.

1.17 La orden awk

La sintaxis de awk esencialmente es la siguiente:

awk [*ops*] '*list_pat_acc*' [*pars*] [*fichero1* [*fichero2*] ...]

donde

ops	son las opciones disponibles en awk (se verán posteriormente).
list_pat_acc	es una lista de patrones/acciones, con el siguiente formato: *patrón {acción}*
pars	se utiliza para pasar parámetros al programa awk, de la siguiente forma: x=... y=...
ficheroX	es un fichero desde donde se leen las líneas que se van a procesar. Si no se indica ningún fichero, se lee la entrada estándar.

El funcionamiento de awk es el mismo que el de sed: la lista especificada de patrones/acciones se aplica sobre cada línea. Es decir sobre cada línea que cumple el patrón, se ejecuta la acción que se especifica a continuación. Ejemplo:

```
awk '/Cristobal/ {print "Cristobal" }' notas
```

Si no se especifica condición, se ejecuta la sentencia para cada línea de la entrada.

```
awk '{print "Otra linea" }' notas
```

1.18 Una pequeña base de datos

Antes de continuar exponiendo detalles sobre esta poderosa herramienta, vamos a presentar el fichero sobre el cual vamos a desarrollar nuestros ejemplos. El fichero contiene información sobre las calificaciones de tres exámenes de un grupo de alumnos. Esta información está dispuesta simplemente en un fichero plano denominado "`notas`", con cada campo separado por el carácter `#`. Es la forma en la que pequeñas bases de datos suelen guardar su información.

Los campos que contiene una línea son los siguientes:

- El nombre de pila del alumno.
- La asignatura a la que se refieren las notas que vienen a continuación.
- A continuación le siguen tres campos con las calificaciones de los tres exámenes que se han realizado en dicha asignatura.

Evidentemente para que este ejemplo fuera más práctico sería necesario hacer algunos arreglos, pero para nuestros objetivos didácticos es un ejemplo bastante ilustrativo.

```
$ tail notas
Santiago#Dibujo#6#7#6
Ana#Fisica#3#6#7
Ana#Matematicas#9#8#9
Ana#Dibujo#6#5#4
Marta#Fisica#2#3#6
Marta#Matematicas#7#6#6
Marta#Dibujo#3#5#4
Pedro#Fisica#6#5#6
Pedro#Matematicas#5#5#9
Pedro#Dibujo#8#5#9
```

1.19 Separación en campos

La herramienta awk supone que la línea (conocida por registro) está dividida en campos separados por un carácter separador, que por defecto es cualquier secuencia no nula de espacios y tabuladores. Para referenciar cada campo existen unas variables awk especiales: $1, $2, ... que indican respectivamente el campo 1, campo 2, etc. La variable $0 se utiliza para referenciar la línea entera.

Existen variables predefinidas en awk que se utilizan para controlar campos y registros:

FS — Separador de campos (Field Separator) de los registros de entrada. Este separador puede configurarse simplemente inicializando esta variable: FS = "*c*", o bien mediante la opción -F*c* de awk como veremos posteriormente.

En el caso en que el separador no sea un espacio, entonces se entiende que existe un campo a cada lado del carácter separador, así por ejemplo en #hola## (suponiendo # como el separador), existen 4 campos.

```
$ awk -F# '{print NF}'
#hola##
4
```

NF — Número de campos (Number of Fields) en el registro de entrada actual.

NR — Número de registros (Number of Records) leídos hasta el momento (la primera línea es el registro 1).

FNR — Número de registros leídos en el fichero actual (es decir esta variable se inicaliza al abrir un nuevo fichero).

RS — Separador de registros (Record Separator) de entrada. Por defecto esta variable es inicializada al carácter salto de línea.

OFS — Separador de campos en salida (Output Field Separator). Este es el carácter que print imprime cuando separa dos cadenas con ",". Inicialmente su valor es un espacio.

ORS — Separador de registros de salida (Output Record Separator). Es el carácter que print añade al final de la cadena que imprime.

1.20 Patrones

La especificación de las líneas sobre las cuales se debe ejecutar una acción awk se realiza mediante patrones, proporcionando una forma muy potente de seleccionar el instante en el que se deben realizar las instrucciones. Los patrones son expresiones regulares al estilo de egrep con algunas características adicionales, y expresiones relacionales. El resto de la sección describe los patrones más usuales.

A. EXPRESIONES REGULARES

- Expresiones regulares al estilo de egrep, encerradas mediante el carácter "/". De esta forma las acciones especificadas se ejecutarán sólo en aquellas líneas que contengan esa expresión regular, es decir alguna cadena que la cumpla. Ejemplo:

```
$ awk '/M/ {print}' notas
$ awk '/^M.*F/' notas
```

B. EXPRESIONES RELACIONALES

- Expresiones relacionales construidas según alguna de las siguientes posibilidades:

  expresión op_relacional expresión

  expresión op_expr_regular expr_regular

  donde

  i. *expresión* puede ser una constante numérica (18, 18., 3.4, 18e2, 1.3E+3...), una constante tira ("Cristobal", "Fisica", "15", ...), una variable predefinida ($0, $1, $2, NF, NR...), una variable definida por el usuario o una función awk.

  ii. *op_relacional* puede ser cualquiera de los seis operadores relacionales de C:

<	menor que
<=	menor o igual que
==	igual que
!=	distinto que
>=	mayor o igual que
>	mayor que

  iii. *op_expr_regular* es ~ o bien !~, operadores que comprueban si la expresión especificada cumple o no, respectivamente, la expresión regular que se indica.

  iv. *expr_regular* es una expresión regular.

Ejemplos:

```
$ awk -F# '$4 > 5 {print $1,$2,$4}' notas
```

El siguiente ejemplo mostraría aquellos registros que empiezan por una J mayúscula. Es decir la información sobre todos los alumnos cuyos nombres empiezan por J.

```
$ awk '$0 ~ /^J.*/ {print $0}' notas
```

El siguiente ejemplo mostrará todas las líneas en las que el primer campo (es decir el nombre del alumno), no contenga una r. Fíjese en que la comprobación no consiste en ver que el primer campo no sea igual a r, sino que no contenga r.

```
$ awk -F# '$1 !~ /r/' notas
```

En el caso de comparaciones con expresiones regulares (y sólo en el caso de nawk), éstas también pueden aparecer entre comillas dobles:

```
$ nawk -F# '$1 !~ "M..t"' notas
```

C. COMBINACIONES BOOLEANAS

- Combinaciones booleanas de expresiones regulares y expresiones relacionales, y paréntesis:

Operador de negación !

! *expr*	Será verdadera si el valor de *expr* es falso, y viceversa.

Operador &&

expr1 && *expr2*	Será verdadera sólo si las dos expresiones son verdaderas.

Operador ||

expr1 || *expr2*	Será verdadera sólo si alguna de las dos expresiones es verdadera.

Operador ( )

(*expr*)	Sirve para agrupar expresiones. Será verdadera si *expr* es verdadera, falso en caso contrario.

Ejemplos:

```
$ awk -F# '$2 ~ /Mat/ && $5 >= 5 {print $1,$5}' notas
```

D. PATRONES BEGIN Y END

BEGIN y END son dos patrones especiales. El primero de ellos hace que la acción especificada sea ejecutada antes de leer la primera línea de entrada.

Por otra parte el patrón END indica que la acción especificada para él, debe ejecutarse justamente después de procesar la última línea de entrada.

Estos dos patrones pueden ser muy útiles, por ejemplo para iniciar alguna variable, o mostrar alguna cabecera que anteceda a la salida, en el caso de BEGIN; y para realizar algún cálculo final, en el caso de END.

Ejemplo:

```
$ awk -F# 'BEGIN {print "Aprobados en Matematicas"}
          $2 ~ /Mat/ && $5 >= 5 {print $1,$5}' notas

$ awk -F# 'BEGIN {print "Aprobados en Matematicas"}
          $2 ~ /Mat/ && $5 >= 5 {print $1,$5}
          END  {print "Presentados: " NR}' notas
```

E. PATRÓN RANGO

Otra forma de especificar líneas en las que ejecutar alguna acción es dar un rango de líneas mediante el carácter ",". La sintaxis en este caso es:

patrón1,patrón2

La orden awk busca la primera ocurrencia de *patrón1* y empieza a ejecutar las acciones correspondientes hasta que aparezca una línea donde se cumpla *patrón2*. Después de esto vuelve a buscar una línea que cumpla *patrón1*.

Algunos ejemplos de rangos son los siguientes:

```
$ awk -F# '/Juan/,/Marta/{print $1,$2,$4}' notas
$ awk -F# '$3 >= 9 && $1~/^[A-C]/,/Marta/' notas
Ana#Matematicas#9#8#9
Ana#Dibujo#6#5#4
Marta#Fisica#2#3#6
```

1.21 Opciones de awk

Veamos cuáles son las opciones posibles para awk en la línea de mandatos, la primera ya ha sido comentada anteriormente y utilizada ampliamente en los ejemplos.

-F c	Define *c* como separador de campos.
-v *var=valor*	Asigna *valor* a la variable *var*. Esta opción puede repetirse varias veces. Es especialmente útil para pasar variables del shell. Esta opción sólo está disponible en nawk.
-f *prog_awk*	La secuencia de patrones/acciones que se quieren ejecutar se encuentran en el fichero *prog_awk*, en vez de en la línea de mandatos. Recuerde no encerrar esta secuencia entre comillas simples en el fichero.

Ejemplos:

```
awk -F# '$1 ~ /Cristobal/ {print $0}' notas
awk -F# '$1 == "Cristobal" {print $0}' notas
awk -F# -v alu=Cristobal 'alu ~$1 {print $1,$5}' notas
```

1.22 Acciones de awk

La parte de acciones de awk se encierra entre dos llaves, y consiste en una lista de cero o más sentencias (con sintaxis muy similar a C), separadas por punto y coma (;), o por saltos de línea. A continuación se presentan las acciones más importantes disponibles.

A. SENTENCIA PRINT, PRINTF

La sintaxis de `print` es:

```
print [expresión] [> fichero]
```

La sentencia `print` escribirá sus argumentos (en el caso de que existan) en la pantalla. La coma entre argumentos se sustituirá por el separador de campos en salida (OFS), y terminará con el separador de registros en salida (ORS). En vez de escribir en pantalla se puede también enviar la salida a un fichero mediante el carácter ">".

```
$ awk -F# '{print $1,$5}'  notas
```

La sentencia `print` proporciona un forma de escribir información en la pantalla muy simple. Para generar una salida formateada es posible utilizar la sentencia `printf`. Esta sentencia interpreta sus argumentos como lo hace la función `printf` del lenguaje C. Observe la siguiente orden para extraer un informe de las calificaciones medias por asignatura.

```
$ awk -F# '{printf "%-15s %4.1f\n", $1, ($3+$4+$5)/3}' notas
```

B. INICIALIZACIÓN DE VARIABLES

Otra de las características de awk la posibilidad de definir variables. Las variables están inicializadas por defecto a la tira nula, en el caso de que se tome como una cadena, y al valor numérico 0 en el caso de tratarse como un número. El usuario siempre puede definir y asignar variables.

Las variables no se definen explícitamente, esto se hace internamente al utilizarlas. La sintaxis de la asignación es la de C:

```
variable = expresión
```

C. CONTROL DE FLUJO

awk dispone de las clásicas sentencias estructuradas de programación para el control de flujo, veamos sus formatos:

```
if (expresión) sentencia [ else sentencia ]
while (expresión) sentencia
do sentencia while (expresión)
for (expr1;expr2;expr3) sentencia
```

Donde la semántica de cada unas de estas instrucciones es análoga a la correspondiente en C. A continuación presentamos algunos ejemplos de su utilización.

El siguiente programa awk calcula la calificación media de todas las puntuaciones del primer examen de Física.

```
$ awk -F# '{if ($2=="Fisica")
                {
                        cont+=$3;
                        num+=1;
                        print $3
                }
        }
      END {print "Media: ", cont/num}' notas
```

Observe que la comprobación de que el segundo campo es igual a `"Fisica"` también se podía haber hecho fuera de la acción, es decir en la parte del patrón.

Un ejemplo con la sentencia `for` para imprimir todos los campos de cada registro de entrada, uno en cada línea:

```
$ awk -F# '$1 == "Juan" {
                        for (i=1; i<=NF; i++)
                                print $i
                }' notas
```

D. OTRAS SENTENCIAS DE CONTROL DE FLUJO

Otras sentencias similares a las existentes en C para el control de flujo son las siguientes:

Sentencia break

Cuando es invocada en el interior de un bucle `for` o `while`, esta sentencia conduce el control de ejecución del programa a la instrucción que sigue después de dicho bucle.

Sentencia continue

Cuando es invocada en el interior de un bucle `for` o `while`, esta sentencia conduce el control de ejecución del programa a la próxima iteración de dicho bucle.

Sentencia exit

Si es invocada desde BEGIN, el programa terminará sin ejecutar nada más (ni siquiera la sección END, si es que ésta existiera).

Si es invocada desde alguna sección diferente a BEGIN o a END, el programa no leerá más líneas de entrada, y pasará directamente a ejecutar la sección END.

Si es invocada desde la sección END el programa terminará en ese momento de ejecución del programa.

Esta sentencia puede devolver un estado pasándolo como argumento.

Sentencia return

La sintaxis de esta instrucción es:

```
return [expr]
```

y su acción es terminar la función desde donde es llamada, devolviendo el valor de *expr*.

Sentencia next

Esta sentencia hace que awk salte inmediatemente al siguiente registro, y que comience a interpretar el programa desde el primer patrón especificado.

1.23 Arrays en awk

Otra característica importante en awk es la existencia de arrays. Los elementos de un array pueden ser o no numéricos y no necesitan ser declarados, su existencia comienza en el momento en el que son mencionados. Ya que no se definen, tampoco está determinado su tamaño, por lo que pueden crecer indefinidamente. Un índice de un array puede ser cualquier valor no nulo, cadenas o números.

Veamos un ejemplo de su utilización. Utilizaremos un array para guardar las notas del primer examen de Física de cada estudiante, y al final las mostraremos en orden inverso al que aparece en el fichero:

```
$ cat reves
nawk -F# '$2=="Fisica" {
                        cont += 1
                        nFi[cont]=$3"<--"$1
                       }
                END {
                        printf "   Calificaciones del\n"
                        printf "   Primer examen de\n"
                        printf "        Fisica\n"
                        printf "-------------------------
---\n"
                        for ( i = cont; i>0; i=i-1 )
                                print nFi[i]
            }' notas
```

La sentencia:

```
for (var in array) sent
```

recorre de forma no predecible *array* haciendo que la variable *var* contenga el índice de cada elemento del array, por lo tanto, la expresión *array[var]* contiene un elemento del mismo en cada iteración.

En nawk existen los arrays multidimensionales, por lo que tienen sentido expresiones como:

```
array[i,j]
```

Sin embargo, en awk no existen tal cual, pero pueden simularse mediante cadenas de caracteres como índices, de la siguiente manera:

```
i=1
j=3
bidi [i",",j]=valor
```

De esta forma el índice del array bidi es la cadena "1,3", con lo cual simularemos una matriz bidimensional mediante un array de una dimensión.

1.24 Otras características de awk

Esta herramienta posee más características interesantes, de las que a continuación enumeramos algunas:

- Están disponibles muchas funciones predefinidas para realizar diferentes tipos de acciones: cálculos numéricos, tratamiento de cadenas de caracteres, etc.
- El usuario puede definir sus propias funciones (sólo en `nawk`).
- La función predefinida `system` permite ejecutar un mandato de la shell desde dentro de un programa `awk`, y continuar en el punto donde fue llamada una vez que dicho mandato ha terminado su ejecución (sólo en `nawk`).

1.25 Pasar información a un programa awk

En numerosas ocasiones es necesario pasar alguna información dentro de un programa awk, para que dicho programa actúe dependiendo del valor de dicha información. Por ejemplo supongamos que queremos obtener las calificaciones de una asignatura. Pero que queremos crear un programa shell que extraiga esta información desde nuestra base de datos. La asignatura que deseamos queremos pasarla como parámetro al programa shell.

El problema que se plantea es el siguiente:

i. Para recoger el valor del parámetro del programa shell, debemos utilizar parámetros posicionales ($1,$2, ...).

ii. En un programa awk estas variables tienen un significado propio.

En el caso de fijar la asignatura, una orden válida podría ser:

```
$ awk -F# '$1 == "Fisica" {print $1,$3,$4,$5}'notas
```

Si queremos hacer variable esta información existen cuatro posibilidades:

1. Utilizar la opción -v:

```
$ cat shellscript
nawk -F# -v asi=$1 '$2 == asi {print $1,$3,$4,$5}' notas
$ shellscript Fisica
```

2. Para pasar una variable en awk se puede usar usando el campo *pars* de la línea de mandatos. Por ejemplo:

```
$ cat shellscript
awk -F#  '$2 == asi {print $1,$3,$4,$5}' asi=$1 notas
$ shellscript Fisica
```

3. Otra posibilidad es dejar fuera de las comillas simples (') aquello que queremos que sea interpretado por la shell:

```
$ cat shellscript
awk -F# '$2 == "'$1'" {print $1,$3,$4,$5}' notas
$ shellscript Fisica
```

4. Por último podemos hacer lo contrario, es decir permitir a la shell que se introduzca en el programa awk, por lo que se debe escapar todo aquello que es específico de awk.

```
$ cat shellscript
awk -F# "\$2 == \"$1\" {print \$1,\$3,\$4,\$5}" notas
$ shellscript Fisica
```

1.26 Interfuncionamiento con otros programas

Otra posibilidad interesante es el uso de awk en conjunción con otros programas.

A lo largo de este curso tendrá sobradas ocasiones de comprobar que las salidad de la mayor parte de los programas son extremadamente sobrias y regulares. Esta característica facilita la construcción de órdenes complejas "ensamblando" otras órdenes.

A continuación se muestran varios ejemplos de utilización de awk. Intente descubrir qué hacen.

```
$ who | awk '$1~/maria/ {print $1, $2}'

$ sort /etc/passwd | awk -F: '$7~/csh/ {print $1,$5}'

$ ls -l | awk 'NR!=1 {count[$3]++ }
                END  {for (own in count)
                          print own,count[own]}'
```

1.27 Prácticas

1. Intente predecir el funcionamiento de las siguientes órdenes, ejecútelas y compare el resultado con su respuesta:

```
$ awk '/M*/' notas
$ awk '/^M*/' notas
$ awk '/M.*/'notas
```

2. Busque la información sobre todos los alumnos excepto la relativa a Marta. Piense tres formas de hacerlo.
3. Visualice desde la línea 2 hasta la 8 del fichero notas. Piense dos formas de hacerlo.
4. Busque cuál es la máxima nota en la asignatura de Física y el alumno que la ha sacado.
5. Saque un informe con las medias de cada uno de los exámenes de cada asignatura.
6. Busque los alumnos que tienen aprobado el primer examen de Matemáticas.
7. Busque los alumnos que tienen aprobado todos los exámenes de Dibujo.
8. Busque los alumnos que tienen aprobado todos los exámenes.
9. Construya una tabla con el número de aprobados (en porcentaje), para cada asignatura y examen, además extraiga porcentajes para todos los exámenes de una misma asignatura, y para cada i-esimo examen de todas las asignaturas.

CAPÍTULO 4

LA SHELL UN LENGUAJE DE PROGRAMACIÓN

1.1 Programas de shell

La shell incluye todo un lenguaje de programación, es decir se suele hablar de programas escritos en shell. El término inglés utilizado para referenciar un programa shell es "script", que al traducirlo al castellano algunos libros llaman guión, o simplemente programa shell o programa de shell (como habitualmente haremos nosotros).

Un programa de shell es simplemente un fichero que contiene una secuencia de mandatos y órdenes, con la misma sintaxis que si fueran ejecutados directamente desde la línea de mandatos. Su ejecución se llevará a cabo de la misma forma que se hace con un programa escrito en un lenguaje de programación secuencial (C, Pascal, etc). Es decir una orden detrás de otra, según aparecen en el fichero, con posibles saltos a funciones, y ejecución de sentencias de control de flujo propias de un lenguaje estructurado.

Existen algunas diferencias según el tipo de shell que interprete un programa de shell. Entre la shell de Bourne y la de Korn, las diferencias son pequeñas, consistiendo esencialemente en mejoras y nuevas posibilidades de la segunda con respecto a la primera. Sin embargo existe una diferencia sustancial entre los lenguajes de programación de la shell de Korn y la shell C.

Debido a la consolidación de la shell de Korn como shell preferida en múltiples entornos, a que la mayor parte de "scripts" están realizados para la shell de Bourne (los cuales correran sin problemas en la shell de Korn), y a las mejoras que aporta esta última, el presente capítulo se centrará esencialmente en la programación en la shell de Bourne, y en el siguiente se comentarán algunas características de la shell de Korn.

1.2 Ejecución de un programa de shell

A. CREAR UN PROGRAMA

Lo primero que se debe hacer cuando se quiere crear un programa shell es editarlo, es decir crear un fichero de texto con algún editor (por ejemplo vi), e introducir en él los mandatos que se quieren ejecutar.

No es necesario compilar este programa shell como ocurre con otros muchos lenguajes, ya que el programa es interpretado instrucción a instrucción por la shell. No por aquella que fue creada al conectarse el usuario, sino por una copia de la misma.

Sin embargo para que este fichero pueda ser ejecutado debe concedérsele permiso de ejecución, para lo cual es necesario ejecutar el mandato:

```
$ chmod u+x programa
```

De esta forma el propietario de este fichero podrá ejecutarlo.

B. INVOCAR A UN PROGRAMA

La forma más usual de ejecutar un programa shell es simplemente invocarlo, es decir, teclear su nombre directamente en la línea de mandatos, acompañándolo o no por una lista de parámetros, según veremos más adelante. Es decir igual que se haría con cualquier orden o mandato primitivo de la shell.

Esto indica que no hay manera de distinguir, desde el punto de vista de su invocación, entre una orden interna de la shell, un programa ejecutable del propio sistema o generado por algún usuario (es decir cuyo contenido es una secuencia de instrucciones para el microprocesador del equipo), o un programa de shell del sistema o generado por un usuario.

Por ejemplo, vamos a hacer un simple y pequeño programa de shell. Para crearlo utilizamos la orden `cat`.

```
$ cat > simple
echo "Contenido del directorio actual"
ls -l | pg
^D
$ chmod u+x simple
$ simple
```

Ya que es posible introducir en un programa de shell cualquier orden que pueda ser ejecutada directamente en la shell, también es válido invocar a otro programa de shell (incluso a él mismo).

C. EJECUCIÓN DE UN PROGRAMA

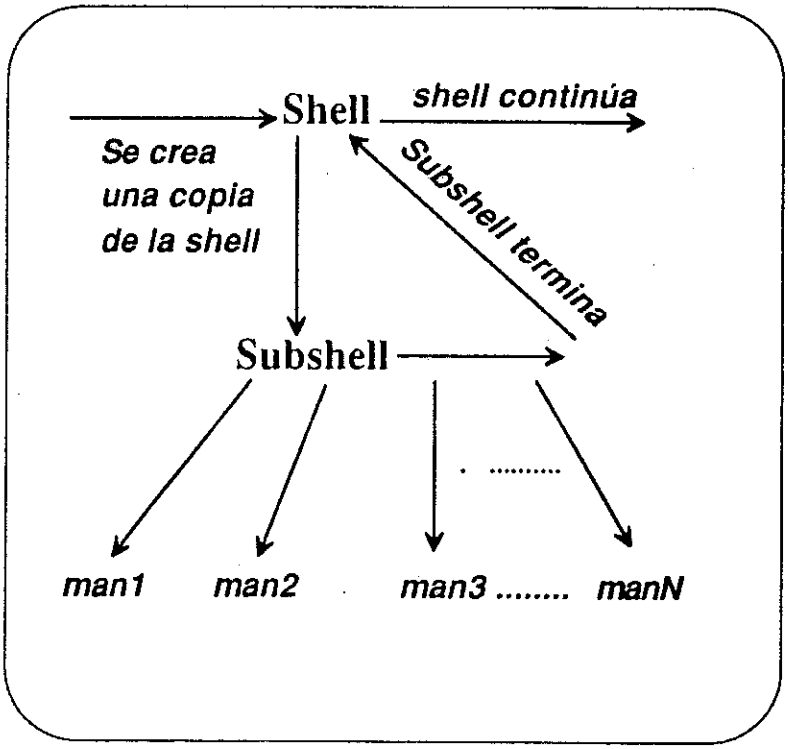
Cuando un programa de shell es invocado directamente en primer plano se siguen los siguientes pasos:

1. La shell busca el programa invocado en la lista de rutas especificadas por la variable de entorno PATH.

2. Si se encuentra, la shell creará una copia de ella misma en memoria. Es esta copia la que se encarga de interpretar y ejecutar cada una de las órdenes que el programa contiene, lo que probablemente conllevará la creación de nuevos procesos que ejecuten cada una de estas órdenes.

3. La shell desde donde se invocó el programa (al igual que ocurre con todos los mandatos que se ejecuten en primer plano), quedará esperando a que éste termine.

4. Una vez que la copia de shell finaliza la ejecución del programa avisará a la shell original, que reanudará su ejecución mostrando de nuevo el indicador al usuario.



Prácticas

- Ejecute y analice la información generada del siguiente, simple pero ilustrativo programa shell:

```
$ cat > ejepshell
ps -f
^D
$ chmod u+x ejepshell
$ ejepshell
```

1.3 Diferentes formas de ejecutar un programa shell

Extisten varias formas de ejecutar un programa shell. Aunque las diferencias entre ellas son bastante sutiles, es importante entenderlas bien puesto que en algunas ocasiones puede ser trascendental la selección apropiada.

A. INVOCACIÓN DIRECTA

La primera de ellas es la que acabamos de ver. Se crea el programa y se le asignan los permisos adecuados de ejecución. Para ejecutarlo basta con teclear su nombre, y ya se ha explicado anteriormente cómo se lleva a cabo su ejecución .

B. UTILIZAR UNA ORDEN SHELL.

Una segunda forma de ejecutar un programa shell es utilizar directamente algunas de las órdenes shell: `sh`, `ksh`, `csh`. Y pasarle como parámetro el nombre del programa shell que se quiere ejecutar (con sus parámetros si es que los tuviese).

Esta orden (la shell misma) activa una subshell que se encarga de ejecutar una a una las órdenes del programa de shell, hasta que,

- no quedan más órdenes que ejecutar,
- la subshell recibe una señal que le haga terminar,
- o se encuentra un error sintáctico en el programa shell.

Funcionalmente este método es igual al anterior, salvo las siguientes diferencias:

1. Se puede seleccionar explícitamente qué shell ejecutará el programa, algo que en el caso anterior viene determinado, como ya veremos, por la shell interactiva desde la que se invoca. Este funcionamiento por defecto puede ser modificado situando algunas instrucciones específicas en el programa de shell.

2. El programa de shell debe tener permiso de lectura, pero no necesita que se le haya asignado el de ejecución.

C. LA ORDEN "."

Cuando se ejecuta un programa de shell mediante alguno de los dos métodos anteriores, cualquier cambio que se realice relativo al entorno de la shell actual, no quedará reflejado en la misma.

Las acciones como cambiar de directorio, crear y asignar variables, definir alias, crear funciones, etc, no quedarán reflejadas en la shell actual, es decir sólo tendrán vigencia mientras que la subshell exista y sólo en ella, desapareciendo sus efectos una vez que ésta termine.

Una forma de condensar una serie de acciones en un fichero "script" y ejecutarlo posteriormente actuando directamente en la shell actual consiste en utilizar la orden ".". Con este método, es la shell actual la que leerá y ejecutará uno a uno los mandatos y órdenes que dicho fichero contiene. El efecto es análogo a teclear estas órdenes directamente en la shell actual.

Veamos un ejemplo. Suponga que se desea una orden que suba dos niveles en el árbol de directorios. Para ello creamos el programa de shell denominado `s2`:

```
$ pwd
/home/ingusr1
$ cat > s2
cd ../..
^D
```

La ejecución de este programa mediante la orden "." hará que efectivamente cambiemos de directorio:

```
$ . s2
$ pwd
/
```

Sin embargo la ejecución mediante la orden `ksh` no tendrá el efecto deseado:

```
$ cd /home/ingusr1
$ ksh s2
$ pwd
/home/ingusr1
```

1.4 Utilizar "( )"

En algunas ocasiones es interesante no escribir directamente los mandatos en un fichero y ejecutarlo posteriormente, pero sin embargo se desea que estos mandatos se ejecuten en una subshell diferente a la actual. La forma de hacer consiste en utilizar el operador "( )".

Para lograr este objetivo se debe situar la lista de órdenes deseadas entre dos paréntesis, separándolas por ";". Esto funciona exactamente igual que colocar estas órdenes en un fichero y ejecutarlo posteriormente.

Los motivos para usar "( )" pueden ser varios:

- La acción que se va a realizar se piensa que no se ejecutará de nuevo, y por tanto no se quiere consumir espacio, pero es necesario crear una subshell para ello. Por ejemplo:

```
$ (cd programas;miprog;ls)
```

- Se quiere redirigir la salida estándar y/o la salida de errores de un conjunto completo de órdenes a un único fichero.

```
$ (cat doc | pr | lp) 2> errores
```

1.5 ¿Qué shell ejecuta un programa?

Dado que existen tres tipos de shell que se pueden estar utilizando en un momento dado, cabe preguntarse en el primer caso de los anterioremente expuestos, cuál es la shell que ejecuta un programa.

En el caso A), el contenido de la primera línea del fichero "script" puede servir para indicar implícita o explícitamente cuál es la shell que lo ejecutará. Sin embargo, esta línea no tendrá efecto (se tomará como un simple comentario) en los casos B) y C). En el caso B) será la shell que se indique explícitamente en la línea de mandatos, mientras que en el C) será la shell interactiva que se esté utilizando en el momento de invocar al programa.

En el caso A), es decir invocar directamente al programa de shell, la primera línea indicará qué shell utilizar. La siguiente tabla informa sobre los posibles contenidos de esta primera línea.

primera línea	shell interactiva	shell que interpreta el programa
#	x	x
no # (:)	sh ó csh	sh
no # (:)	ksh	ksh
#!*ruta_shell*	x	*ruta_shell*

Esta tabla indica que en el caso de que la primera línea (en la primera columna) haya un carácter # (que no es seguido por !), la shell que interpreta el programa de shell es la shell interactiva que está utilizando el usuario.

En el caso de que este primer carácter no sea #,de Korn si ésta es la shell interactiva. Se suele poner ":" como primer carácter del fichero, ya que este carácter es una sentencia de la shell que no hace nada, y siempre se ejecuta correctamente. De esta forma se elimina la posibilidad de escribir por descuido un comentario en la primera línea .

Hay que tener en cuenta que el resultado que se ofrece en la tabla es el análisis de todos los casos, pero que estos resultados se producen como consecuencia del siguiente convenio (que es quizás lo que suele aparecer con frecuencia en los libros). Bajo la shell C, todos los programas de shell serán ejecutados por la shell de Bourne, salvo que el primer carácter del fichero sea "#". Como consecuencia, cuando el primer carácter es #, la shell que interpreta un programa es la propia shell interactiva.

La última posibilidad es indicar explícitamente cuál es la shell que debe ejecutar el programa. Para ello la primera línea del fichero debe ser de esta forma:

```
#!ruta_shell
```

Por ejemplo:

```
#!/usr/bin/ksh
```

En realidad aquí podría indicarse cualquier programa que fuese capaz de interpretar el fichero.

Prácticas

1. Cree el siguiente programa de shell:

```
$ cat > queshell
ps -f
^D
```

2. Asígnele permiso de ejecución.

3. ¿Qué shell debe interpretar el programa?

4. Ejecútelo y analice la información que devuelve.

5. Añada una línea al principio del fichero con el carácter "#". Ejecute el programa desde los tres tipos de shell y analice el resultado. Para activar una shell como interactiva, basta con invocar la orden correspondiente: `sh`, `csh`, `ksh`.

1.6 Parámetros posicionales

En la shell se definen unos parámetros especiales denotados por números. Mediante los parámetros posicionales un programa de shell puede acceder a los argumentos que se le pasan en el momento de su invocación.

Los parámetros `$1`, `$2`, `$3`, `$4`, ... hacen referencia al argumento primero, segundo, tercero, cuarto, ... que se le hayan pasado al programa cuando se invocó.

El parámetro `$0` contiene el nombre del programa de shell. Así en el siguiente ejemplo:

```
$ cat > parpos
echo $0
echo $1
echo $2
echo $3
^D
$ chmod u+x parpos
$ parpos Hola que tal
parpos
Hola
que
tal
```

A. ASIGNACIÓN DE LOS PARÁMETROS POSICIONALES

Cada copia de shell tiene su propio conjunto de parámetros posicionales. El mecanismo de pasar argumentos a un programa de shell supone realmente asignar valores a los parámetros posicionales de la copia de shell que se crea.

Es posible asignar valores a los parámetros posicionales de la shell actual, pero no directamente. Para hacer esto, en sh y ksh se debe utilizar la orden `set` que permitirá asignar valores al conjunto de parámetros posicionales, aunque no individualmente.

Ejemplo, ejecute las órdenes siguientes :

```
$ set Hola que tal
$ echo $1
Hola
$ echo $2
que
$ echo $3
tal
```

Otro ejemplo:

```
$ set `ls`
```

El resultado es la asignación de los nombres de ficheros y directorios del directorio actual a los parámetros posicionales.

B. PARÁMETROS POSICIONALES ALTOS

En la shell de Bourne sólo se puede hacer referencia a 9 parámetros posicionales en cada momento. Para acceder al décimo, úndecimo argumento, ..., se utiliza la instrucción `shift` de la que hablaremos posteriormente.

Sin embargo, en la shell de Korn sí es posible acceder a parámetros posicionales por encima del 9, simplemente encerrando entre llaves el número del parámetro:

```
$ echo ${10}
$ echo ${12}
```

Prácticas

1. Conteste antes de ejecutar cuál sería el resultado de los siguientes mandatos, bajo sh o ksh:

```
$ set 1 2 3 4 5 6 7 8 9 0 1 2 3
$ echo $10
$ echo $12
```

2. ¿Qué ocurre si bajo sh se ejecuta lo siguiente:?

```
$ set 1 2 3 4 5 6 7 8 9 0 1 2 3
$ echo ${10}
```

1.7 Variables "$"

Para trabajar con los parámetros posicionales la shell nos ofrece variables especiales que empiezan por el carácter "$". Éstas son las siguientes:

$#	Esta variable contiene el número de parámetros posicionales existentes (sin contar $0).
$*	Esta variable contiene la lista de parámetros posicionales desde 1 en adelante.

Veamos algunos ejemplos:

```
$ set a b c d
$ echo $#
4
$ echo $*
a b c d
$ set "a    b"    c   d
$ echo $#
3
$ echo $*
a    b c d
```

1.8 Asignación y acceso a variables de shell

La instrucción de asinación de variables en el lenguaje de programación shell es igual que cuando se inicializa una variable interactivamente. Su sintaxis es similar a la de otros lenguajes de programación (recuerde no dejar espacios antes ni después del operador de asignación, y de encerrar entre comillas dobles *valor* si esta cadena contiene espacios o tabuladores).

```
var=valor
```

Cuando se quiere acceder al valor de una variable se utiliza el operador "$". La shell sustituirá la variable por su valor. Existen diferentes posibilidades para acceder al valor de una variable como veremos a continuación.

A. OPERADOR { }

El operador { } se utiliza para diferenciar caracteres que componen el nombre de la variable de aquellos que no forman parte de él. Un ejemplo ha sido ya visto al tratar con parámetros posicionales. Veamos otro ejemplo:

```
$ dia=dia
$ echo "Buenos " $dias
```

La intención de este ejemplo es utilizar la variable `dia` que ha sido inicializada con la cadena `dia`, para posteriormente formar la cadena `dias`. Pero ya que la "s" última es un carácter válido para formar nombres de variables, la shell interpretará que se quiere acceder a la variable `dias`. Puesto que esta variable no está definida, su valor será nulo, imprimiéndose sólo la cadena `"Buenos "`, que no era lo deseado.

Para separar por tanto aquello que forma parte del nombre de lo que no, se utiliza el operador { }.

```
$ echo "Buenos " ${dia}s
```

B. UTILIZAR UN VALOR POR DEFECTO

No definida o valor nulo

Cuando una variable no está definida, o lo está pero contiene un valor nulo, puede interesar tomar un valor por defecto. Esto se consigue mediante la siguiente expresión.

```
${var:-valordefecto}
```

Esta expresión devuelve el contenido de *var* si está definida y tiene un valor no nulo. Por ejemplo si la variable `total` inicialmente no está definida:

```
$ echo ${total}


$ echo "El resultado es: ${total:-0}"
El resultado es: 0
$ total=1
$ echo "El resultado es: ${total:-0}"
El resultado es: 1
```

Se puede acceder a los parámetros posicionales de esta manera:

```
${$1:-0}
```

No definida

En algunas ocasiones interesa utilizar un valor por defecto sólo en el caso de que la variable no esté definida. La expresión que se utiliza para ello es ligeramente distinta a la anterior:

```
${var-valordefecto}
```

C. UTILIZAR OTRO VALOR

No definida o valor nulo

Existe una expresión opuesta a la anterior. En este caso, si la variable está definida y contiene un valor no nulo, entonces en vez de usarse dicho valor, se utiliza el que se especifica. En caso contrario, el valor de la expresión es la cadena nula. La expresión para esto es:

```
${var:+valordefecto}
```

Por ejemplo:

```
$ total=25
$ echo ${total:+30}
30
$ total=
$ echo ${total:+30}

$
```

No definida

En algunas ocasiones puede también interesar que el comportamiento anterior sólo suceda cuando la variable no esté definida. La expresión para ello es:

```
${var+valordefecto}
```

D. ASIGNAR UN VALOR POR DEFECTO

No definida o valor nulo

En otras ocasiones interesa no sólo utilizar un valor por defecto, sino asignarlo. La expresión que se utiliza para ello es la siguiente:

```
${var:=valordefecto}
```

Si el contenido de `var` es no nulo, esta expresión devuelve dicho valor. Si el valor es nulo o la variable no está definida entonces el valor de la expresión es `valordefecto`, el cual será también asignado a la variable `var`. Veamos un ejemplo en el que se supone que la variable `total` no está definida:

```
echo ${total}


$ echo "El resultado es: ${total:=0}"
El resultado es: 0
$ echo ${total}
0
```

Los parámetros posicionales **NO** se pueden asignar de esta forma.

No definida

En algunas ocasiones puede interesar también utilizar y asignar un valor por defecto sólo en el caso de que la variable no esté definida. La expresión ahora es:

```
${var=valordefecto}
```

E. IMPRIMIR UN MENSAJE DE ERROR

No definida o valor nulo

En otras ocasiones no interesa utilizar ningún valor por defecto, sino comprobar que la variable está definida y contiene un valor no nulo. En este último caso interesa avisar con un mensaje y que el programa de shell termine. La expresión para hacer esto es:

```
${var:?Mensaje}
```

Por ejemplo:

```
$ total2=${total:?"variable no valida"}

total: variable no valida
```

En el caso de que la variable total no esté definida o contenga un valor nulo, se mostrará el mensaje especificado en pantalla, y si esta instrucción se ejecuta desde un programa de shell, éste finalizará.

No definida

En algunas ocasiones puede también interesar que el comportamiento anterior sólo suceda cuando la variable no esté definida. La expresión para ello es:

```
${var?mensaje}
```

1.9 Lectura desde la entrada estándar (`read`)

La orden `read` permite asignar valores proporcionados por la entrada estándar a variables de shell. Es el método de "input" que ofrece la shell como lenguaje de programación. El contenido con el que se asignan las variables puede ser suministrado por un usuario del programa de shell o por un programa.

La forma más simple de leer una variable es la siguiente:

```
read variable
```

Después de esto, el programa quedará esperando hasta que se le proporcione una cadena de caracteres terminada por un salto de línea. El valor que se le asigna a *variable* es esta cadena (sin el salto de línea final).

La instrucción `read` también puede leer simultáneamente varias variables:

```
read var1 var2 var3 var4 ... varN
```

La cadena suministrada por el usuario empieza por el primer carácter tecleado hasta el salto de línea final. Esta línea se supone dividida en campos por el separador de campos. Este separador es definido por la variable de shell IFS (Internal Field Separator) que por defecto es una secuencia de espacios y tabuladores.

El primer campo tecleado por el usuario será asignado a *var1*, el segundo a *var2*, etc. Si el número de campos es mayor que el de variables, entonces la última variable contiene los campos que sobran. Si por el contrario el número de campos es menor que el de variables, las variables que sobran tendrán un valor nulo.

La shell de Korn proporciona dos características más para la instrucción `read`. Una de ellas es la variable por defecto REPLY, que es utilizada por defecto en el caso de que no se especifique ninguna en la invocación de `read`.

La segunda característica es la posibilidad de combinar con la propia orden `read`, la impresión de un mensaje solicitando la entrada de uno o más valores. Si en la primera variable de esta instrucción aparece un carácter "?", todo lo que reste hasta el final de este primer argumento de `read`, se considerará el mensaje que se quiere enviar a la salida estándar.

Ejemplo:

```
read nom?"Nombre y dos apellidos? " ap1 ap2
Nombre y dos apellidos? Rafael Gomez Lopez
```

1.10 La orden `test`

La orden `test` está pensada para ser utilizada con la sentencias de control de flujo como pronto veremos. Evalúa una expresión y devuelve el resultado como el código de retorno de la orden. La sintaxis de la instrucción es:

```
test expr
```

El primer ejemplo que vamos a ver es consultar si una variable está o no definida:

```
test $var
```

Si la variable *var* está definida entonces el resultado de la instrucción anterior será 0 (correcto), si no lo está, será un valor distinto de 0. Recuerde que este valor se puede obtener accediendo a la variable $? inmediatamente después de ejecutar esta instrucción.

La instrucción `test` tiene otras muchas posibilidades, por ejemplo puede consultar la existencia o no de algún fichero, sus permisos, etc. Así, para preguntar si un fichero existe y es regular (no especial):

```
$ test -f fich
$ echo $?
```

Si esta variable tiene un valor diferente de 0, el fichero no existe, o existe pero no es regular.

En muchas ocasiones, alguno de los argumentos que se le pasa a la instrucción `test` es el contenido de una variable (`$var`). Puede ser que este contenido sea nulo, por lo que al sustituir este valor en la expresión, la instrucción `test` quede sin algún argumento. Esta situación provoca un error y la finalización del programa de shell.

La opción `-n` de `test` comprueba si la cadena que se le pasa es o no vacía. Pruebe el siguiente ejemplo:

```
$ cad=
$ test -n $cad
$ test -n "$cad"
```

Existe otra sintaxis para esta sentencia, que ofrece un mejor aspecto a la hora de combinarla con sentencias de control. Consiste en eliminar la palabra "test", y encerrar todo lo demás entre corchetes. Por ejemplo:

```
[ $VAR ]
```

o

```
[ -f fich ]
```

Es importante respetar todos los espacios que aquí aparecen.

A continuación se listan varias de las comprobaciones que puede realizar esta sentencia:

-f *fich*	existe el fichero y es un fichero regular
-r *fich*	existe el fichero y es de lectura
-w *fich*	existe el fichero y es de escritura
-x *fich*	existe el fichero y es ejecutable
-h *fich*	existe el fichero y es un enlace simbólico.
-d *fich*	existe el fichero y es un directorio
-p *fich*	existe el fichero y es una pipe con nombre
-c *fich*	existe el fichero y es un fichero especial de caracteres
-b *fich*	existe el fichero y es un fichero especial de bloques
-u *fich*	existe el fichero y está puesto el bit SUID
-g *fich*	existe el fichero y está puesto el bit SGID
-s *fich*	existe el fichero y su longitud es mayor que 0
-z s1	la longitud de la cadena s1 es cero
-n s1	la longitud de la cadena s1 es distinta de cero
s1 = *s2*	la cadena s1 y la s2 son iguales
s1 != *s2*	la cadena s1 y la s2 son distintas
s1	la cadena s1 es nula NO NULA
n1 -eq *n2*	los enteros n1 y n2 son algebraicamente iguales.
n1 -ne *n2*	los enteros n1 y n2 no son iguales.
n1 -gt *n2*	n1 es estrictamente mayor que n2.
n1 -ge *n2*	n1 es mayor o igual que n2.
n1 -lt *n2*	n1 es menor estricto que n2.
n1 -le *n2*	n1 es menor o igual que n2.

Las expresiones primarias pueden combinarse con los siguientes operadores:

!	operador unario de negación
-a	operador AND binario
-o	operador OR binario

Por ejemplo:

```
$ test 0 -lt 0 -o -n "No nula"
$ echo $?
```

1.11 Las sentencias `true` y `false`

Existen dos sentencias que no hacen nada y que devuelven verdadero y falso respectivamente, estas son `true` y `false`.

Compruébelo mediante las siguientes órdenes:

```
$ true
echo $?
0
$ false
echo $?
1
```

Realemente tanto `true` como `false` son dos alias, definidos a partir de las instrucciones que de verdad realizan la acción comentada.

Un lugar donde se suelen utilizar estas sentencias es en la implementación de bucles indefinidos.

1.12 Estructuras de control

A. SENTENCIA IF

La shell dispone de la sentencia `if` de bifurcación de la ejecución de un programa, imprescindible en todo lenguaje de programación. La estructura sintáctica más simple de esta sentencia es:

```
if lista_mandatos
then
    lista_mandatos
fi
```

En la parte reservada para la condición de la sentencia `if`, debe aparecer una lista de mandatos separados por ";". Cada uno de estos mandatos es ejecutado en el orden en el que aparecen. La condición para `if` tomará el valor de salida del último mandato ejecutado.

Como condición se puede poner cualquier mandato que interese, pero lo más habitual es utilizar diferentes formas de la orden `test`. Por ejemplo:

```
if [ -f midoc ]
then echo "midoc existe y es un fichero regular"
fi
```

Para dirigir el control de ejecución del programa hacia otra secuencia de mandatos, en caso de que la condición no resulte ser verdadera, la shell dispone de la siguiente estructura:

```
if lista_mandatos
then
    lista_mandatos
else
    lista_mandatos
fi
```

Por ejemplo:

```
if [ -f "$1" ]
then
    pr $1
else
    echo "$1 no es un fichero regular"
fi
```

La combinación `else if` es posible, y para ello existe la palabra reservada `elif`.

Ejemplo:

```
if [ -f "$1" ]
then
    lp $1
elif [-d "$1" ]
then
    lp $1/*
else
    echo "No es posible imprimir $1"
fi
```

B. SENTENCIA WHILE

La sentencia `while` tiene la estructura sintáctica siguiente:

```
while lista_mandatos1
do
    lista_mandatos2
done
```

La secuencia de mandatos en el interior del bucle `while` (*`lista_mandatos2`*) será ejecutada mientras que *`lista_mandatos1`* devuelva verdadero (lo que significa que el última mandato en esta lista devuelva verdadero).

```
while [ ${resp:=s} = s ]
{
    # Ejecutar alguna accion
    .....
    read resp?"Quiere usted continuar(s/n)? "
}
```

C. SENTENCIA UNTIL

La sentencia `until` es el complemento de la sentencia `while`. Su formato es el siguiente:

```
until lista_mandatos1
do
    lista_mandatos2
done
```

Aquí se ejecutará la secuencia de mandatos en el interior del bucle hasta que se cumpla la condición, es decir hasta que el último mandato de *lista_mandatos1* sea verdadero, o lo que es lo mismo "mientras" no se cumpla la condición.

Por ejemplo, en el caso anterior, cuando solicitamos una respuesta para continuar o no, el usuario podría introducir por descuido un salto de línea. Nos interesaría volver a solicitar la respuesta en esta situación:

```
while [ ${resp:=s} = s ]
{
    # Ejecutar alguna accion
    .....
    read resp?"Quiere usted continuar(s/n)? "
    until [ "$resp" != "" ]
    do
        read resp
    done
}
```

D. SENTENCIA FOR

En muchas ocasiones es necesario ejecutar una misma tarea para varios ficheros (o cadenas en general, por ejemplo sobre los parámetros posicionales). La sentencia `for` soluciona este problema, su estructura es:

```
for var in lista_palabras
do
    lista_mandatos
done
```

lista_palabras es una cadena formada por campos, separados unos de otros por espacios y tabuladores. En cada iteración del bucle la variable *var* se inicializa con el siguiente campo, y se ejecuta la secuencia de mandatos *lista_mandatos*.

El siguiente ejemplo muestra como crear una copia de todos los fuentes C:

```
for i in `find /home/proy -name "*.c" -print`
do
    cp $i $i.BAK
done
```

Si se omite "in *lista_palabras*" en la sentencia, se entiende que la cadena que se recorrerá es la formada por los parámetros posicionales.

COMILLAS } || → | \ - ejecuta su valor sustituyendo el lugar que ocupa por la salida standard de la orden

E. SENTENCIAS BREAK Y CONTINUE

La sentencia `break` se utiliza para terminar la ejecución de un bucle `while` o `for`. Es decir el control continuará por la siguiente instrucción del bucle. Si existen varios bucles anidados, `break` terminará la ejecución del más próximo (del más interno).

Es posible salir de *n* niveles de bucle mediante la instrucción `break` *n*.

La instrucción `continue` desvía el control a la próxima iteración de un bucle, con lo cual dejará de ejecutar cualquier sentencia posterior de la iteración en curso.

F. SENTENCIA CASE

Esta sentencia permite controlar ramificaciones múltiples. El formato básico de esta sentencia es el siguiente:

```
case cadena in
    patrón1) lista_mandatos1;;
    patrón2) lista_mandatos2;;
    .....
    patrónN) lista_mandatosN;;
esac
```

La shell comprueba si *cadena* cumple alguno de los patrones especificados. Esta comprobación es realizada en orden, es decir empezando por *patrón1* y terminando por *patrónN*. En el momento en que se detecte que la cadena cumple algún patrón, se ejecutará la secuencia de mandatos correspondiente hasta llegar a ";;". Estos dos puntos y comas fuerzan a salir de la sentencia `case` y a continuar por la siguiente sentencia después de `esac`.

Las reglas para componer patrones son las mismas que para formar nombres de ficheros, así por ejemplo, el carácter "*" es cumplido por cualquier cadena, por lo que suele colocarse este patrón en el último lugar, actuando como acción por defecto. Ejemplo:

```
# visualizar un fichero
if [ -f "$1" ]
then
    vi $1
else
    echo "El fichero $1 no existe"
    read resp?"Desea crearlo? "
    case $resp in
        [yY]*)vi $1
              ;;
        [nN]*)echo "Adios"
              ;;
        *)    echo "Respuesta incorrecta"
              ;;
    esac
fi
```

1.13 La sentencia `shift`

La sentencia `shift` desplaza los parámetros posicionales hacia la izquierda un número especificado de posiciones. La sintaxis para la sentencia `shift` es:

```
$ shift n
```

donde *n* es el número de posiciones a desplazar. El valor por defecto para *n* es 1. En el caso de la shell de Bourne en la que sólo es posible acceder directamente hasta el noveno parámetro posicional, ésta es la forma de obtener parámetros posicionales altos.

Correspondencia de argumentos con los parámetros posicionales.

man	arg1	arg2	arg3	arg4	....	argN
|	|	|	|	|	.....	|
$0	$1	$2	$3	$4	....	$N

Después de ejecutar la sentencia `shift` con argumento 1, los parámetros posicionales se desplazarán un lugar a la izquierda. Esto significa que se pierde el primer argumento:

Correspondencia de argumentos con los parámetros posicionales después de shift

man	arg1	arg2	arg3	arg4	.....	argN
|		|	|	|	.....	|
$0		$1	$2	$3	.....	$N-1

El siguiente ejemplo muestra cómo acceder al décimo argumento:

```
$ cat > decimo
shift 9
echo "El decimo argumento es: " $1
^D
$ decimo `ls /`
```

Uno de los lugares donde más se utiliza `shift` es en el recorrido de una lista de palabras. Por ejemplo:

```
$ cat > buscarcad
cadena=$1
shift
while [ "$*" != "" ]
do
    grep $cadena $1
    shift
done
^D
$ buscarcad errno *.h
```

1.14 Los operadores && y ||

A. OPERADOR &&

La shell dispone de dos operadores condicionales más: el operador AND (&&), y el OR (||). La sintaxis del primero es:

```
mandato1 && mandato2
```

El funcionamiento de esta expresión es la siguiente, se ejecutará *mandato1* y si devuelve un estado de salida correcto, entonces (y sólo entonces) se ejecutará *mandato2*. Esta expresión devolverá el estado correcto si los dos mandatos se ejecutan correctamente, y falso si el primer mandato o el segundo devuelve falso.

Este operador puede sustitirse por una sentencia `if` de la siguiente forma:

```
if mandato1
then
    mandato2
fi
```

Prácticas

1. Analice y pruebe el siguiente ejemplo:

```
$ [ $VAR ] && grep "$VAR" *
```

B. OPERADOR ||

La sintaxis de este operador es:

```
mandato1 || mandato2
```

El funcionamiento es el siguiente, se ejecuta *mandato1* y si devuelve correcto entonces no se ejecutará *mandato2*. Si el resultado de la primera ejecución fuera falso entonces sí se ejecutaría la segunda.

Existe también una forma equivalente a este operador mediante la sentencia `if`.

```
if !mandato1
then
    mandato2
fi
```

Prácticas

1. Analice y pruebe el siguiente ejemplo:

```
$ grep Sistema doc || echo "No se ha encontrado Sistema"
```

1.15 El mandato `expr`

Esta orden toma sus argumentos los evalúa e imprime el resultado. Proporciona un mecanismo para la evaluación de expresiones aritméticas. Todos los términos de la expresión deben separarse por espacios, y tiene el inconveniente de que todos los metacaracteres de la shell deben escaparse mediante \ Por ejemplo el operador de multiplicación (*) es un metacarácter, por lo que para que la shell no lo interprete, debe escaparse.

Ejemplos:

```
$ expr 2-1
1
$ expr 3 / 2
1
$ expr 3 \* 2
6
```

El resultado impreso puede recogerse y asignarse a una variable:

```
a=`expr $b + 1`
```

A continuación las operaciones más importantes que se pueden realizar con esta sentencia.

`expr1 | expr2`	devuelve el resultado de la primera expresión si no es nulo ni 0, en caso contrario devuelve el resultado de la segunda.
`expr1 \& expr2`	devuelve `expr1` si ninguna de las dos expresiones es nula o 0, en caso contrario devuelve 0.
`expr1 = expr2`	devuelve 1 si las dos expresiones (enteros o cadenas) son iguales, y 0 en caso contrario.
`expr1 \> expr2`	devuelve el resultado de la comparación entera (mayor estricto que) si los argumentos son enteros, en caso contrario realiza una comparación léxicografica.
`expr1 \>= expr2`	igual que la anterior, pero ahora se comprueba si es `expr1` es mayor o igual que `expr2`.
`expr1 \< expr2`	menor estricto que.
`expr1 \<= expr2`	menor o igual que.

`expr1 != expr2`	distinto que.
`expr1 {+, -} expr2`	Suma y resta de los dos argumentos enteros.
`expr1 {\*, /, %} expr2`	Multiplicación, división y resto de dos argumentos enteros.
`expr1 : expr2`	Se comprueba si el `expr1` cumple la expresión regular `expr2`. Las expresiones regulares que se utilizan son las mismas que en `sed`.

1.16 Terminar un programa de shell (`exit`)

Todos los mandatos y órdenes tienen un estado de finalización, que por convenio es 0 cuando la ejecución terminó correctamente, y un valor distinto de 0 cuando lo hace anormalmente.

Si un programa de shell termina sin errores devolverá correcto, pero también es posible devolver explícitamente un valor mediante la sentencia `exit`. La ejecución de esta sentencia finaliza en ese instante el programa de shell, devolviendo el valor que se le pase como argumento, o el estado de la última orden ejecutada si no se le pasa ningún valor.

Ejemplo:

```
if grep "$1" `find $2 -type f -print` > /dev/null
then
    exit 0
else
    exit 10
fi
```

1.17 Creación de procesos desde un programa de shell

A. INVOCACIÓN EN PRIMER PLANO

Desde un programa de shell es posible invocar a otro programa de shell. Este segundo programa se ejecutará evidentemente en otro proceso independiente, mientras que el primero esperará a que termine.

Prácticas

1. Analice detenidamente y ejecute el siguiente programa de shell, que sirve para ilustrar la creación de procesos que tiene lugar cuando se ejecuta un programa de shell desde otro.

```
$ cat otrop
varexp=`expr ${varexp:-0} + 1`
export varexp
echo $varexp
ps -f
[ $varexp -eq 5 ] || otrop
```

B. INVOCACIÓN EN SEGUNDO PLANO

Una forma de ejecutar desde un programa de shell una orden u otro programa de shell es activarlo en segundo plano, es decir utilizando el operador &.

La ejecución de esta orden se realizará de la misma forma que los procesos en segundo plano activados directamente desde la shell interactiva. Es decir, la orden activada se ejecutará independientemente del programa de shell, que continuará su ejecución sin esperar a que termine.

Prácticas

1. Analice y compare el siguiente ejemplo con el del apartado anterior.

```
$ cat otropsu
varexp=`expr ${varexp:-0} + 1`
export varexp
echo $varexp
ps -f
[ $varexp -eq 40 ] || (otropsu &)
```

1.18 Las variables $$ y $!

Existen dos variables más en la shell que empiezan por `$`. Estas variables contienen identificadores de procesos.

La variable `$$` contiene el identificador del proceso que está ejecutando el programa de shell.

Así, por ejemplo:

```
$ cat idps
ps
echo $$
^D
$ idps
   PID TTY      TIME COMD
  4953 term/3    0:00 ksh
  4954 term/3    0:00 ps
  1493 term/3    0:01 ksh
4953
```

Debido a que los números de procesos son únicos, este valor suele utilizarse para construir nombres de ficheros temporales. Estos ficheros nomalmente se suelen situar en directorios específicos para temporales: `/temp`, `/usr/temp`, `/var/temp`, etc.

Ejemplo:

```
mifich=/temp/tp$$
....
rm -f $mifich
```

Otra variable interesante que proporciona la shell es `$!`, que contiene el identificador del último proceso que se ejecutó en segundo plano.

Este dato es interesante si luego se quiere realizar alguna señalización a los procesos arrancados en segundo plano. Por ejemplo se pueden guardar los identificadores de estos procesos, y cuando interese matarlos.

```
$ cat varda
dormir &
id1=$!
dormir &
id2=$!
ps -f
sleep 3
kill -9 $id1 $id2
```

1.19 Tratar opciones (getopts)

Muchas órdenes y programas existentes en el sistema Unix particularizan su ejecución mediante opciones que se teclean en el momento de la invocación. Además a menudo los argumentos varían dependiendo de la opción seleccionada.

Existe una orden que facilita esta tarea: `getopts`. Esta facilidad ayuda a la captura de opciones, y está basada en las reglas estándares para la sintaxis de las opciones en Unix:

- Las opciones están formadas por un guión más una letra minúscula o mayúscula inmediatamente después.
- El orden de las opciones es indiferente, pero suelen aparecer antes de los argumentos propiamente.
- Cada opción puede llevar uno o más argumentos.
- Las opciones sin argumentos pueden agruparse después de un único guión.

La sintaxis de `getopts` es la siguiente:

```
getopts cadena_opciones variable [args ...]
```

En *`cadena_opciones`* se sitúan las opciones válidas del programa. Cada letra en esta cadena significa una opción válida. El carácter ":" después de una letra indica que esa opción lleva un argumento asociado (o grupo de argumentos separados por una secuencia de espacios y tabuladores).

Esta orden se combina con la sentencia `while` a fin de que itere una vez por cada opción que aparezca en la línea de mandatos. En *`variable`* se almacena cada vez dicha opción, y el índice del próximo argumento que se va a procesar se guarda en la variable OPTIND. Esta variable se inicializa a 1 siempre que se invoque un programa de shell.

Cuando a una opción le acompaña un argumento, `getopts` lo sitúa en la variable OPTARG.

Si en la línea de mandatos aparece una opción no válida entonces asignará "?" a *`variable`*.

Mediante *`[args ...]`* se puede hacer que `getopts` busque las opciones en otra cadena en vez de hacerlo en la secuencia de parámetros posicionales.

Veamos un ejemplo:

```
while getopts hi:a op
do
    case $op
    in
        i) echo "Opción i, argumento $OPTARG"
           # Realizar acción asociada a -i
           ......
           ;;
        a) echo "Opción a"
           # Realizar acción asociada a -a
           .....
           ;;
        h) echo "Opción h"
           # Realizar acción asociada a -h
           .....
           ;;
        ?) echo "Use: traops [-i fichero][-a][-h]  "
           # Se ha tecleado una opción inválida
           ;;
    esac
done
shift `expr $OPTIND - 1`
echo "el resto de argumentos: $*"
```

Es decir después de la sentencia `shift`, $1 tendrá el primer argumento propiamente (no opción) de la línea de mandatos.

1.20 La sentencia `eval`

La orden `eval` toma una serie de cadenas como argumentos:

```
eval [arg1 [arg2] ...]
```

El comportamiento de eval es el siguiente. La shell expande los argumentos de `eval` como de costumbre. La shell forma entonces una cadena al concatenar los argumentos de `eval` separándolos por un espacio. Finalmente lee y procesa la cadena resultante como haría sobre cualquier mandato. Por tanto el resultado es un doble procesamiento de los argumentos de `eval`.

Esta instrucción se debería utilizar cuando:

- Se quiera reexaminar el resultado de una expansión realizada por la shell.
- Se quiera encontrar el valor de una variable cuyo nombre es el valor de otra variable.
- Ejecutar una línea que se ha leído o compuesto internamente en el programa de shell.

Prácticas

1. Conseguir el último parámetro posicional

   ```
   eval ultimo='${#}
   ```

2. Ejecute y compare los siguientes ejemplos:

   ```
   var=HOME
   echo '$'$var
   eval echo '$'$var
   ```

3. Construya un cauce de dos mandatos en una variable, y luego intente ejecutarlo.

4. Lea un mandato desde teclado y ejecútelo.

1.21 Prácticas

1. Construya un programa de shell para ordenar alfabéticamente los argumentos que se le pasen.
2. S_06_06_05_1 Construya un programa de shell que genere los primeros 1000 números enteros.
3. S_06_06_05_2 Construya un programa de shell que lea un fichero y lo muestre con los números de líneas a la izquierda.
4. S_06_06_05_3 Construya un programa de shell que tome como argumento una lista de ficheros y directorios y saque un mensaje (fichero regular, sin/con permiso de lectura, ...) sobre el tipo y los permisos de cada uno de ellos.
5. S_06_06_05_4 Realice un programa de shell que muestre todos los ficheros de texto en el directorio que se especifique (el actual si no se indica). La opción -s no mostrará ninguno de los ficheros de texto, sólo devolverá un estado dependiendo de si hay alguno o no.
6. S_06_06_05_5 Construya un programa para ser insertado en `/etc/profile` el cual descubrirá si el usuario que se está conectando es el especificado en el primer parámetro posicional, y si es así enviará un mensaje al administrador informando de ello.
7. Lo mismo que antes pero ahora la conexión se realiza por una línea especificada.

CAPÍTULO 5

CARACTERÍSTICAS DE LA PROGRAMACIÓN EN SHELL KORN

1.1 El mandato `select`

El el capítulo anterior se ha visto la programación en shell, fijándonos sobre todo en las características de la shell de Bourne, las cuales son válidas en la shell de Korn. En este capítulo vamos a ver algunas de las novedades incorporadas en esta última shell.

Vamos a empezar con un mandato que permite, con un mínimo esfuerzo, visualizar una lista de alternativas para que el usuario seleccione una de ellas: es decir, la construcción de un menú. El formato de `select` es el siguiente:

```
select variable [ in lista_palabras ]
do
    lista_mandatos
done
```

La forma de trabajar `select` es la siguiente:

- La ksh muestra las palabras de *`lista_palabras`* en una o más columnas en el error estándar. Cada uno de estas palabras va precedida por un número (existe la posibilidad de formatear la salida mediante las variables `LINES` y `COLUMNS`).
- Una vez que se han mostrado en pantalla todas las opciones, se visualiza el contenido de la variable PS3. El usuario debería inicializarla con un mensaje que solicite la selección de una opción.
- Una vez que el usuario del programa selecciona una opción, la shell sitúa este número en la variable `REPLY`, y la palabra asociada en *`variable`*.

- La ksh ejecuta *lista_mandatos* para cada selección que se realice hasta que encuentra una sentencia `break`, `return` o `exit`.

Veamos un ejemplo, (es importante indicar que debe ser ejecutado con ksh):

```
#!/usr/bin/ksh
# Ejemplo de select
PS3="Introduzca una opción (q para salir) :"
tput clear
while true
do
    select var in `find . -name "*.o" -print`
    do
        case $REPLY
        in
            q*|Q*) exit;;
            *[!0-9]*) echo "Opción incorrecta";;
        esac
        rm -i $var
        break
    done
done
```

Prácticas

1. Piense por qué ha sido conveniente introducir un bucle `while` en el ejemplo anterior, aunque `select` ya lo hace por sí mismo.

1.2 Atributos de variables

La shell de Korn permite asignar atributos a una variable. Por ejemplo es posible definir un tipo entero para una variable, o una variable cadena que debe ser justificada a la izquierda. El mandato utilizado por ksh para esta asignación de atributos es `typeset`.

A. CONVERTIR A MAYÚSCULAS

Asignando el atributo `u` a una variable, la ksh cambiará todas las letras minúsculas a mayúsculas cuando acceda a su valor. Por ejemplo:

```
$ typeset -u may="hola que tal"
$ echo $may
HOLA QUE TAL
```

B. CONVERTIR A MINÚSCULAS

Contrariamente a lo anterior, el atributo `-l` cambiará a minúsculas.

```
$ typeset -l min="HOLA QUE TAL"
$ echo $min
hola que tal
```

C. VARIABLES ENTERAS

Mediante el atributo `i` se pueden definir variables enteras. La shell Korn evaluará el valor que se le asigna a una variable entera, y guardará el resultado como un número binario en vez de guardarlo como una cadena. Las operaciones aritméticas realizadas con este tipo de variables pueden tardar menos tiempo.

```
$ typeset -i a=80
```

El alias `integer` (`integer="typeset -i"`) puede ser igualmente utilizado para definir variables enteras:

```
$ integer a=80
```

Las variables enteras pueden utilizarse en expresiones aritméticas sin necesidad de acceder a ellas mediante el operador $.

D. VARIABLES JUSTIFICADAS

Mediante los atributos `L` y `R` pueden definirse variables justificadas a la izquierda o a la derecha, respectivamente.

```
$ typeset -L8 nombre
```

Esta definición fuerza a la ksh a justificar a la izquierda el valor de la variable `nombre`, ajunstándolo a 8 caracteres. Si la longitud del valor es más pequeño que 8, entonces la ksh rellenará con espacios. Si es mayor que 8, la ksh truncará la cadena, descartando los caracteres que sobren a la derecha.

```
$ nombre="Pedro"
$ echo "$nombre es mi nombre"
Pedro    es mi nombre
$ nombre=Margarita
$ echo "$nombre es su nombre"
Margarit es su nombre
```

De forma análoga cuando se asigna el atributo `R` a una variable, su valor es justificado a la derecha ajustándolo al tamaño especificado. La ksh rellenará con espacios a la izquierda si la longitud del valor es menor que este tamaño, y cortará por la izquierda si es mayor.

```
$ typeset -R8 apellidos
$ echo "$apellidos"
   Lopez
$ apellidos="Rodriguez"
$ echo "$apellidos"
odriguez
```

E. SÓLO LECTURA

Mediante el atributo -r se pueden definir variables de sólo lectura:

```
$ typeset -r tasa=36
tasa=38
ksh: tasa: is read only
```

F. EXPORTAR UNA VARIABLE

La shell de Korn puede exportar variables a procesos hijos mediante la misma sintaxis que la shell de Bourne:

```
$ varexp="Cadena"
$ export varexp
```

pero también puede hacerlo mediante la sentencia `typeset`:

```
$ typeset -x varexp
```

1.3 Arrays

La shell de Korn permite también utilizar arrays de una dimensión. La sintaxis es:

```
identificador[subindice]
```

El subíndice debe ser una expresión aritmética cuyo rango comienza en 0 y termina en un valor que depende de la implementación particular.

No es necesario declarar un array, una vez que se referencia cualquier variable con un subíndice válido, será creado automáticamente. Existe sin embargo la posibilidad de definir un array (si se conoce su tamaño) y dar atributos a todos los elementos del mismo. Para hacer esto se utiliza la sentencia `typeset`.

```
typeset [atributos] identificador[tamaño]
typeset -i datos[10]
```

Si se referencia un array sin subíndice, se entiende que se está accediendo al primer elemento de ese array.

Para referenciar a todos los elementos de un array se debe utilizar el carácter "*" como subíndice.

```
dato[*]
```

Ejemplo:

```
typeset -i aleatorio[10]
while [ ${cont:=0} -lt 10 ]
do
    aleatorio[$cont]=$RANDOM
    cont=`expr $cont + 1`
done
echo ${aleatorio[*]}
```

1.4 Notaciones especiales

La shell permite notaciones especiales para algunas expresiones. En algunos casos su utilización puede mejorar las prestaciones de las operaciones que se quieran realizar.

A. SENTENCIA LET

La sentencia `let` evalúa las expresiones aritméticas que se le pasen como argumentos. La ksh evalúa cada argumento que se le pase como una expresión aritmética independiente. Por tanto los argumentos deberán separarse entre sí encerrándolos entre simple o dobles comillas, si fuera necesario.

Esto puede ser necesario puesto que muchos de los operadores aritméticos coinciden con metacaracteres de la shell. Por este motivo se ofrece otra notación alternativa `((...))`, equivalente a `let "..."`. Es decir lo que se encuentra dentro de los dobles paréntesis se supone encerrado entre dos comillas dobles.

```
let x=x+1 "a=a*1"
```

El valor retornado por `let` es 0 (verdadero) si el valor de la última expresión evaluada no es cero, y 1 (falso) en caso contrario. De esta forma, `((...))` se utiliza para evaluar condiciones:

```
let a=${1-10}
while (((a=a-1) >= 0)) && read -r linea
do
    eval $linea
done
```

B. NUEVA SUSTITUCIÓN DE MANDATOS

La shell de Korn permite una nueva notación para la sustitución de mandatos tradicional: (` `), de la siguiente manera:

```
`mandato` = $(mandato)
```

Por ejemplo:

```
for i in $(find . -name "*.c" -print)
do
    vi $i
done
```

C. SUSTITUCIÓN NUMÉRICA

También la shell de Korn dispone de una nueva notación para situar en la línea de mandatos el resultado de la evaluación de una expresión aritmética, esta notación es:

```
$((expresión))
```

que es equivalente a

```
`expr expresión`
```

La diferencia entre esta notación y `let` es la siguiente: las dos evalúan la expresión que se les pasa, pero `let` no devuelve el valor de dicha evaluación, (como mandato devuelve verdadero o falso), y por lo tanto no puede ser utilizada para forma parte de una línea de mandatos.

Así por ejemplo:

```
x=3
echo "El incremento de x es $((x+1))"
4
```

En esta orden, se evalúa la expresión `x+1`, y el resultado se sustituye por `$((x+1))`. Posteriormente la orden `echo` mostrará el resultado. También se podría haber asignado este valor a la variable x, y posteriormente mostrar el valor:

```
echo "El incremento de x es $((x=x+1))"
4
echo $x
4
```

Esta notación es equivalente a la correspondiente con `expr` funcionalmente, pero la proporcionada por la shell de Korn es bastante más eficiente.

CAPÍTULO 6

INTRODUCCIÓN AL LENGUAJE C

1.1 Introducción

El lenguaje de programación C está íntimamente ligado al sistema Unix. Su evolución y aceptación han ido creciendo paralelamente al desarrollo y utilización de este sistema operativo.

El lenguaje C fue diseñado e implementado por Dennis Ritchie en 1972 en los Laboratorios Bell de AT&T, basándose en un lenguaje ya existente, B, creado por Johnson y Kernighan en 1973, descendiente a su vez del lenguaje BCPL (Richards 1969).

El primer desarrollo de Unix fue realizado en lenguaje ensamblador, pero en 1973, Ritchie y Thompson reescribieron el núcleo del sistema Unix en lenguaje C, lo que rompía con todos los esquemas ya que que la mayoría de los sistemas operativos estaban escritos en lenguaje ensamblador.

Hoy en día más del 90% del código de Unix está escrito en C, reservando para el lenguaje ensamblador sólo aquellas partes que requieran una ejecución muy crítica.

Actualmente C se está extendiendo a diferentes entornos académicos, computacionales, científicos, comerciales, de gestión, etc. Su popularidad está en parte ligada a la fuerte expansión del sistema Unix, pero sólo en parte, ya que actualmente existen compiladores de C para una amplia variedad de equipos, desde micros hasta mainframes.

Evidentemente la aparición de espectaculares compiladores en el mundo de los ordenadores personales (integrados en todo un entorno de trabajo, junto con editores con múltiples características, depuradores, opciones de configuración de la memoria, etc), ha potenciado de manera decisiva su aceptación y accesibilidad.

Por otra parte el éxito de este lenguaje de programación va lógicamente acompañado por importantes características como son las siguientes:

- Estructuración, C ofrece las clásicas sentencias de programación estructurada, y por otra parte dispone de sentencias que permiten un código menos estructurado pero quizás en algunos casos más eficiente.
- Modularidad, facilidad para dividir programas en múltiples módulos.
- Flexibilidad, gracias a lo cual C puede utilizarse para desarrollar aplicaciones de muy diversa índole.
- Portabilidad, incorpora mecanismos que facilitan la independencia de las aplicaciones del sistema en el que finalmente se ejecuten.
- Alto/bajo nivel, en C es posible escribir aplicaciones de muy alto nivel, o bien escribir código muy cercano al hardware de la máquina (acceso a registros, rutinas de interrupción, etc).
- Múltiples bibliotecas de funciones implementadas, donde en parte radica su potencia.

El sistema Unix ofrece varias herramientas para soportar el desarrollo de aplicaciones en C. El lenguaje por sí solo, y la combinación de este lenguaje conjuntamente con las llamadas al sistema que Unix proporciona constituyen un solución válida para la creación de programas.

Por todo ello no podíamos exponer el Entorno de Desarrollo en Unix sin presentar el lenguaje de programación C. En modo alguno este capítulo pretende explicar C, sino tan sólo mostrar lo mínimo necesario para poder entender los ejemplos que aparecerán en los próximos capítulos.

1.2 Apariencia sintáctica de un programa

Sobre el siguiente ejemplo se van a discutir los principales elementos que componen un programa C. El programa solicita líneas (una cadena de caracteres terminada con un carácter salto de línea), las cuales almacena en un array unidimensional, y posteriormente las imprime cambiando las letras minúsculas a mayúsculas.

El ejemplo es sencillo y no tiene interés salvo el de ilustrar un programa C.

```
/*****************************************************************/
/*              Convertir líneas a mayúsculas                    */
/*****************************************************************/
#include "stdio.h"
#define MAXLINEA 81     /* Longitud máxima de la línea
                           (uno más para el carácter nulo) */

#define CAR_NULO '\0'   /* Byte nulo */

char linea[MAXLINEA];   /* Array donde se guardara una linea */
main ()
{
char resp;
void amayu();

    do
    {
          printf ("Introduzca una linea (max. %d cars)\n",
                        MAXLINEA-1);
          gets(linea);
          amayu();
          printf ("Desea continuar (s/n)? ");
          while ((resp=getchar()) == '\n');
    }
    while (resp != 'N' && resp != 'n');
}

void amayu()
{
int i;

    for (i=0; linea[i] != CAR_NULO; i++)
    {
          putchar(toupper(linea[i]));
    }
    putchar('\n');
}
```

A. ESQUEMA DE UN PROGRAMA

Un programa C está formado por uno o más ficheros de texto. Cada uno de ellos está compuesto por una secuencia de funciones con el siguiente formato:

```
tipo nombre_función(argumentos)
definición_de_argumentos;
{
definición_de_variable locales;

    sentencias;

}
```

El lenguaje C no es como otros lenguajes, en los que la descripción de todas las funciones y procedimientos pueden estar descritos internamente en la primera función (por ejemplo Pascal).

Antes de la primera función en el fichero o bien entre funciones (aunque no es muy normal), pueden aparecer instrucciones para el preprocesador, definiciones de variables, declaración de nuevos tipos de datos, etc.

Cualquier texto encerrado entre /* y */ se considerará un comentario, es decir, algo que no será analizado por el compilador, y que sólo tiene significado para el programador.

B. PREPROCESADOR

Incluir ficheros

Antes de que actúe el compilador, un programa denominado preprocesador realiza unas acciones básicas iniciales: lee el fichero ejecutando instrucciones que comiencen por el carácter "#" y además quita los comentarios. Este carácter debe estar en la primera columna puesto que en caso contrario, el preprocesador no hará caso a dicha instrucción.

Habitualmente el preprocesador se encuentra integrado con el compilador, es decir que se ejecutará automáticamente cuando se manda compilar un programa.

Así, en el ejemplo la primera instrucción que aparece para el preprocesador es:

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

Esta instrucción indica al preprocesador que, en el lugar donde aparece, inserte el contenido del fichero que se indica entre ángulos "< >". En este caso el fichero `stdio.h` contiene las declaraciones correspondientes a las funciones estándares de entrada/salida de C.

Esta instrucción se utiliza habitualmente para insertar ficheros que contienen declaraciones de variables, de nuevos tipos, de funciones, etc, de aquí la extensión `.h` ("header", cabecera).

Aunque el preprocesador inserta cualquier tipo de fichero texto, es una buena costumbre de programación NO utilizar este procedimiento para incluir ficheros con código C. El código estará descrito en otros ficheros C, o en alguna biblioteca (como ocurre en el caso de `stdio.h`).

Definir macros

Otra acción muy común realizada por el preprocesador es la de definir macros. En la mayoría de los casos definir un nombre para una cadena de carateres o alguna constante como en el ejemplo:

```
#define MAXLINEA        81
```

o incluso dar argumentos para estos nombres definiendo por tanto macros.

En el caso del ejemplo, el preprocesador sustituirá cualquier aparición de `MAXLINEA` por su valor (81). Realmente casi cualquier aparición, puesto que en algunos casos no procede realizar esta sustitución, por ejemplo si este nombre apareciera dentro de una cadena encerrada por comillas.

C. EJECUCIÓN DEL PROGRAMA

Todo programa C debe tener una única función denominada `main` que como su nombre indica es de fundamental importancia. Es la función por donde comienza a ejecutarse el programa.

Salvo que el programa se interrumpa, la ejecución empezará por la primera sentencia de `main` y terminará por la última.

Esta función principal posiblemente llamará a otras, las cuales a su vez utilizarán otras funciones, etc. El anidamiento de funciones se produce por tanto en el momento de ejecución del programa, y no desde el punto de vista de su situación en el fichero, donde cada función es independiente.

1.3 Tipos de datos

Como todo lenguaje de programación, C dispone de tipos de datos, algunos son básicos y otros compuestos, es decir, inductivamente formados a partir de los simples y otros compuestos.

La sintaxis para definir una variable de un determinado tipo es la siguiente:

```
tipo id1, id2, ....;
```

En este apartado vamos a enumerarlos brevemente.

A. CARACTERES

Son elementos del conjunto de caracteres, habitualmente ASCII. Para definir una variable de tipo carácter:

```
char car;
```

En muchos programas puede extrañar que para tratar información de tipo carácter se utilicen variables de tipo entero. El motivo es que los caracteres son realmente almacenados como enteros, es decir un carácter se guarda como su código dentro del conjunto de caracteres que se esté usando. Por tanto es posible también tratar numéricamente una variable de tipo `char`. Su rango va desde -128 a 127.

Además muchas funciones que tratan con caracteres, devuelven un entero en situaciones especiales (por ejemplo -1), lo que obliga a tratar con enteros.

Si se necesita trabajar con un carácter como un byte de memoria, la mejor definición de tipo es `unsigned char`. Desde el punto de vista numérico el rango de este tipo es 0-255 (es decir no incluye números negativos), y es la definición apropiada cuando las operaciones que se van a realizar sobre la variable son de byte más que de carácter (desplazamientos, máscaras, etc).

B. ENTEROS

Existen tres tipos de variables enteras: `int`, `short` y `long`. Los tres tipos son enteros pero ocupan diferentes tamaños en memoria, por lo que tienen diferentes rangos. La utilización de uno u otro es una decisión del programador.

En lo que se refiere a los tamaños lo único que se asegura es la siguiente relación:

```
tam_short <= tam_int <= tam_long
```

Situaciones habituales que se suelen encontrar son:

tam_short	tam_int	tam_long
16	16	32
16	32	32

Ejemplos de definición de variables enteras:

```
int var, suma, resto;
short cont;
long i, maximo;
```

Las variables aquí definidas almacenarán un entero con signo, es decir que pueden contener números negativos. Sin embargo, si sólo se quiere trabajar con números no negativos, o por ejemplo se quiere trabajar con la variable entera como un campo de bytes, se pueden utilizar los tipos `unsigned short`, `unsigned int` o `unsigned long`, dependiendo del tamaño necesitado.

C. VARIABLES PUNTO FLOTANTE

Los tipos `float` y `double` son los que C ofrece para trabajar con números reales. Toda la aritmética en punto flotante es realizada en doble precisión, lo cual significa un coste para promocionar todas las variables `float` a `double` antes de realizar las operaciones, y el proceso inverso una vez que el resultado se almacene en variables de tipo `float`.

```
float tasa, integral;
double masa, energia;
```

D. VALORES BOOLEANOS

El lenguaje C no dispone de tipos booleanos explícitamente, sino que utiliza valores enteros. El convenio que se sigue es que valores diferentes de cero se interpretan como verdadero, y 0 significa falso.

Habitualmente se realizan las siguientes definiciones para el preprocesador:

```
#define TRUE    1
#define FALSE   0
```

E. PUNTEROS

Un puntero es una variable que apunta a determinado tipo de datos, es decir el contenido de esta variable es la dirección de memoria donde comienza dicho tipo. La sintaxis para definir un puntero es la siguiente:

```
tipo *nombre_ptr;
```

Algunas características de los punteros son las siguientes:

- Una variable puntero puede asignarse. Los valores que deben asignárseles son lógicamente direcciones de memoria. Esto se consigue, bien solicitando memoria libre al sistema (mediante funciones específicas para ello), o bien asignándole la dirección de algún objeto ya definido.
- Es quizás uno de los errores más frecuente de la programación en C, utilizar un puntero que todavía no ha sido asignado. Esto puede ocasionar situaciones desastrosas posiblemente al intentar escribir en direcciones de memoria sin permiso.
- La forma de acceder al contenido referenciado por un puntero es preceder el nombre de la variable puntero mediante el operador "*":

  ```
  *nombre_ptr;
  ```

Esta expresión es manejada exactamente igual que una variable de tipo `tipo`.

F. ARRAYS

Como habitualmente en la mayoría de los lenguajes de programación un array es un tipo compuesto, formado por una sucesión de objetos del mismo tipo. La definición de un array multidimensional es:

```
tipo nombre_array [d1][d2] ... [dn];
```

Cada uno de los elementos del array llamado *nombre_array* es de tipo *tipo*, y *d1*, *d2*, ..., *dn*, son las simensiones del array. Cada uno de los índices del array varía entre 0 y d_i-1.

Algunas características importantes de los arrays son:

- Los elementos del array se almacenan consecutivamente en memoria.
- El nombre del array es la dirección del mismo, es decir la dirección de la primera posición de memoria que ocupa.
- El nombre del array es una dirección constante, es decir no puede modificarse su valor (como se haría con un puntero).

La sintaxis para acceder a los elementos de un array es:

```
nombre_array [i][j] ... [k]
```

G. ESTRUCTURAS

Una estructura es el mismo concepto que el de registro en otros lenguajes como Pascal o Ada. Los elementos de una estructura pueden ser de cualquier tipo a diferencia de un array. Cada uno de los elementos de una estructura es un campo de la misma.

La sintaxis para definir una estructura es:

```
struct
{
     tipo1 campo1;
     tipo2 campo2;
     .......
} nombre_estructura;
```

La forma de acceder a un campo de la estructura es:

```
nombre_estructura.campoi
```

H. UNIONES

Una unión es similar a una estructura excepto que únicamente uno de sus componentes se usa en cada instante. Es decir, en cada momento sólo tiene sentido acceder a un campo de la unión, lo cual es responsabilidad del programador. A todos los componentes de una unión se le asigna el mismo espacio de memoria. Por tanto a la unión se le asignará el mayor tamaño ocupado por sus componentes.

La definición de una unión es la siguiente:

```
union
{
    tipo1 campo1;
    tipo2 campo2;
    ......
}nombre_unión;
```

La forma de acceder a un campo de una unión es la misma que en el caso de una estructura:

```
nombre_unión.campoi
```

1.4 Construcciones más comunes

En este apartado simplemente vamos a enumerar algunas de las construcciones más comunes:

A. ASIGNACIÓN DE VARIABLES

La asignación tiene la forma siguiente:

```
variable = expresión ;
```

La ejecución de esta sentencia consiste en la evaluación de *expresión*, y la asignación del resultado a *variable*.

Esta sentencia admite asignación múltiple, ya que en realidad *variable=expresión* es una expresión que devuelve el valor asignado a la variable, y se convierte en sentencia al añadir el carácter ";" final. Por tanto es válido:

```
var1 = var2 = var3 = expresión ;
```

cuyo resultado es la asignación de la evaluación de la expresión a *var3*, *var2* y *var1* y en este orden.

B. SENTENCIA IF

Esta es la clásica sentencia de salto condicional. Como es habitual, se disponen de varias combinaciones:

```
if (expresión)
    sentencia
```

o bien con las dos ramificaciones:

```
if (expresión)
    sentencia
else
    sentencia
```

Donde `sentencia` puede ser una sentencia simple:

```
sentencia_simple;
```

o bien una sentencia compuesta, es decir formada por una lista de sentencias (simples y/o compuestas) por lo que habría que utilizar llaves para encerrar dicha lista:

```
{
    lista_sentencias;
}
```

C. BUCLE WHILE

La sintaxis del bucle `while` es:

```
while (expresión)
    sentencia
```

donde análogamente la sentencia puede ser simple o compuesta. La semántica de esta construcción es la lógica: se ejecutará `sentencia` mientras que expresión se evalúe como verdadero.

D. BUCLE DO-WHILE

La sintaxis de este bucle es:

```
do
    sentencia
while (expresión);
```

Mientras que *expresión* resulte verdadero se ejecutará *sentencia*. La diferencia con el bucle anterior es que la condición se comprueba al final, con lo cual el bucle se ejecutará al menos una vez.

E. BUCLE FOR

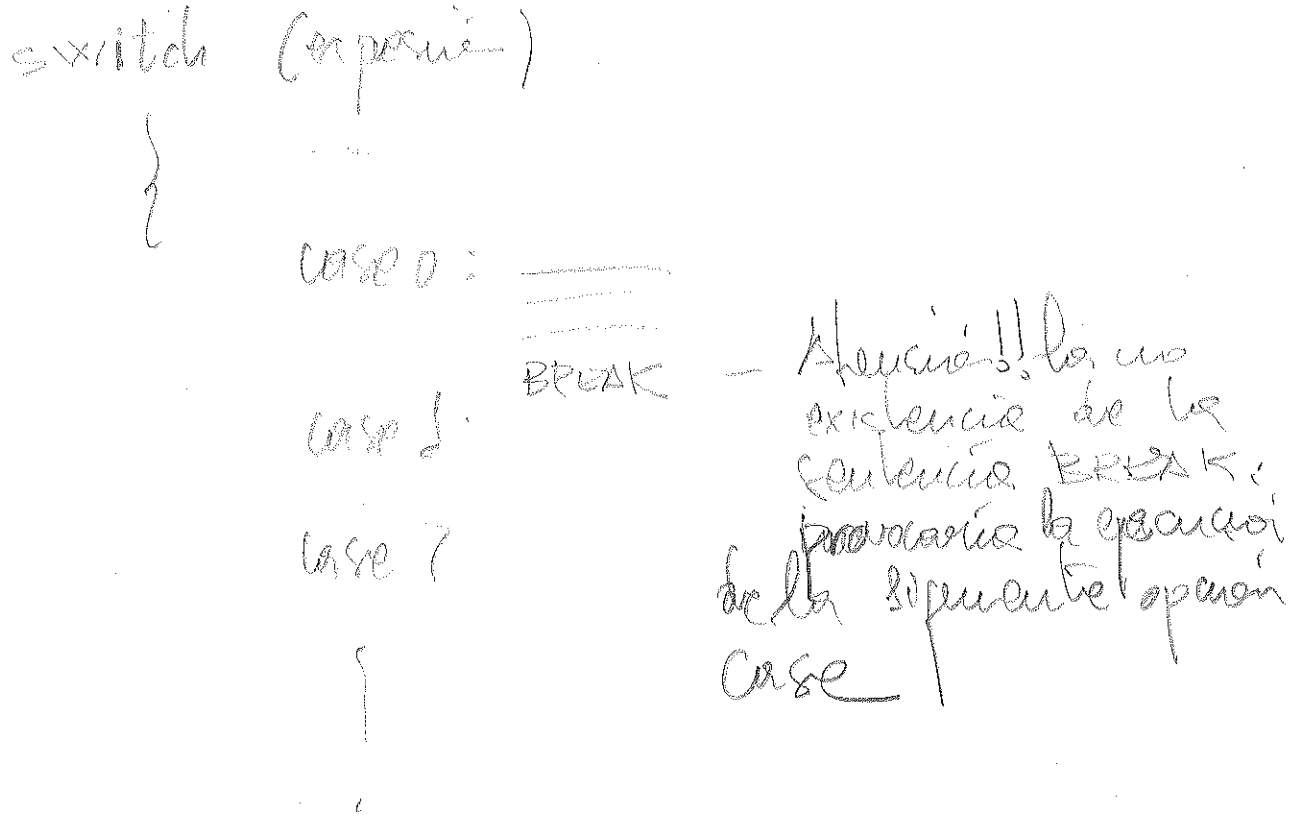
La sintaxis es:

```
for (expr1; expr2; expr3)
    sentencia
```

La semántica de esta construcción puede darse aproximadamente en términos de la del bucle while de la siguiente manera:

```
expr1;
while (expr2)
{
    sentencia
    expr3;
}
```

(El uso de la sentencia `continue`, no explicada en este curso, puede hacer falsa esta equivalencia).



1.5 C de Kernighan-Ritchie y ANSI C

Dennis Ritchie fue quien diseñó e implementó el lenguaje de programación C. El libro escrito por B.W. Kernighan y D.M. Ritchie "*The C Programming Language*". Englewood Cliffs, N.J., Prentice-Hall, ha servido como un estándar en los últimos tiempos. Este estándar es conocido popularmente como el C de Kernighan-Ritchie o el C *del libro*.

Durante los últimos años la popularidad del lenguaje C ha crecido intensamente. El instituto americano ANSI (American National Standards Institute) ha desarrollado un estándar del lenguaje C. Este estándar abarca temas como el entorno de ejecución de un programa, la sintaxis y semántica del lenguaje, bibliotecas y ficheros cabecera.

A. UNIX SVV4 Y ANSI C

En la versión SVV4 de AT&T de Unix se ha incorporado el sistema de compilación C Edición 5.0, que permite compilación ANSI C aunque no obligatoriamente. En versiones de SVV4 la compilación se supondrá que implícitamente es en modo ANSI.

Vamos a presentar dos de las cuestiones introducidas en el estándar ANSI.

B. PROTOTIPOS DE FUNCIONES

El principal cambio entre ANSI C y versiones anteriores es la técnica de prototipo de funciones introducida en la primera. Una función debe definirse describiendo los tipos y los nombres de sus argumentos directamente entre los paréntesis de la función:

```
tipo nombre_función(tipo1 arg1, tipo2 arg2,...,tipon argn)
```

y se declara describiendo los tipos de cada uno de los argumentos de la función:

```
tipo nombre_función(tipo1, tipo2,...,tipon);
```

Un prototipo de función como el anterior proporciona información para comprobar que cada llamada a la función que se realiza cumple con los tipos de todos sus argumentos. Además cuando el tipo de un argumento es compatible con el tipo utilizado en la llamada a la función el compilador realizará la conversión oportuna.

C. CARACTERES MULTIBYTE

Para internacionalizar el lenguaje C se han definido caracteres multibyte. Es decir, en algunos idiomas existe un enorme conjunto de caracteres que evidentemente no es posible representar con sólo un byte. Para ello ANSI C permite la utilización de secuencias de bytes en vez de bytes simples para representar caracteres especiales.

CAPÍTULO 7
COMPILACIÓN DE C

1.1 Fases en la compilación de C

El proceso de compilación del lenguaje de programación C es análogo al de otros lenguajes imperativos. La compilación de todo programa C tiene como fin último la generación de un fichero que sea ejecutable, bien en el sistema donde se compila, bien en otro sistema distinto (a esto se le llama un compilador cruzado).

Antes de obtener este ejecutable es posible generar ficheros intermedios que corresponde a alguna de las fases del proceso de compilación. Estas fases son las siguientes:

A. PREPROCESAMIENTO

Como ya hemos comentado en el capítulo anterior, antes de que tenga lugar la compilación propiamente de un programa C, el preprocesador se encarga de realizar algunas acciones previas de poca complejidad. Básicamente expansión de macros, inclusión de ficheros, etc.

Este procesamiento previo no generará ningún fichero resultado de estas acciones, salvo que se solicite con alguna opción.

Habitualmente el preprocesador es invocado automáticamente por el propio compilador, de forma que se ejecutará automáticamente al compilar un programa.

B. GENERACIÓN DE CÓDIGO OBJETO

La siguiente fase es propiamente la de compilación de un programa. Esta fase es realmente la más compleja de todas, (la cual a su vez está dividida en varios pasos, transparentes al usuario).

El objetivo de la compilación es el de generar un fichero con código máquina, un fichero objeto. Este fichero es equivalente semánticamente (realiza las mismas acciones) que el programa fuente C que se compila, pero escrito en el lenguaje especifico de la máquina (instrucciones del microprocesador).

Un fichero objeto no es directamente ejecutable en el sistema, es decir no tiene todo el código necesario para hacerlo. Normalmente este fichero objeto contendrá llamadas a funciones, cuyo código no se encuentra en él mismo sino en otro fichero objeto, o utilizará variables cuya dirección asignada se encuentra en otro fichero.

C. EDICIÓN DE ENLACES DE UNO O MÁS FICHEROS OBJETO

El último paso que se ejecutará es la generación del fichero ejecutable. Esta acción se lleva a cabo "enlazando" los ficheros objetos y bibliotecas que componen todo el programa.

La tarea de enlazar esencialmente consiste en la resolución de referencias externas, es decir el sustituir las llamadas a funciones o variables externas (definidas en otro fichero distinto al que lo usa), por sus direcciones de memoria.

1.2 cc: Compilador Unix de C

El compilador estándar de Unix, habitualmente integrado en el propio sistema operativo, se denomina `cc`. Se utiliza e invoca igual que un mandato de shell, es decir dispone de opciones para particularizar la ejecución del compilador y argumentos. Su sintaxis es la siguiente:

```
$ cc [opciones] [fich1 [fich2] ... ]
```

A. TIPOS DE FICHEROS

Los tipos de ficheros implicados en una compilación en Unix son los siguientes:

`.c`	Los ficheros que terminen en ".c" son ficheros de texto que contienen un programa en lenguaje C.
`.i`	Los ficheros generados por el preprocesador.
`.o`	Los ficheros terminados en ".o" son ficheros objeto.
`.s`	Los ficheros terminados en ".s" son ficheros escritos en código ensamblador.
`a.out`	Los nombres de los ficheros ejecutables no tienen ninguna restricción, salvo las comunes para los nombres de ficheros en Unix. En el caso de que no se indique ninguno, el mandato `cc` elige por defecto `a.out`.

El compilador cc permite realizar cada una de las fases que antes hemos descrito, separadamente o de forma automática. El mecanismo para realizar una fase u otra es lógicamente la utilización de las opciones disponibles.

B. INVOCAR AL PREPROCESADOR

Automáticamente, cuando se solicita generar un fichero objeto o directamente un ejecutable, a partir de un fichero fuente C, el mandato cc invoca al preprocesador. Si se quiere obtener en un fichero (.i) el resultado de este proprocesamiento se debe utilizar la opción -P.

Prácticas

Obtenga y analice el resultado de pasar el preprocesador a un fichero fuente.

cc
opción función

C. OBTENER UN FICHERO OBJETO

El mandato `cc` genera un fichero objeto (`.o`) a partir de un fichero fuente C (`.c`). La sinitaxis para hacer esto es:

```
$ cc -c [fich1.c [fich2.c] ... ]
```

La opción `-c` indica traducir a ficheros objeto cada uno de los ficheros fuentes C que se le pasan como argumento. Después de este mandato, en el directorio actual se habrán situado los ficheros objetos generados.

D. GENERAR UN EJECUTABLE

Ésta es la opción por defecto del mandato cc, es decir, si se invoca sin opciones intentará generar un ejecutable. Si algunos de los ficheros que se le pasan como argumento es un fichero fuente C lo traducirá a un fichero objeto, pero sin generarlo explícitamente en el directorio. Si ya es un fichero objeto, lógicamente no tendrá que realizar niguna traducción.

Finalmente generará un fichero ejecutable a partir de los ficheros objetos que se le hayan pasado explícitamente, y de las traducciones implícitas que haya realizado de los fuentes C.

Lógicamente no debe existir ningún error en el programa, y entre todos los ficheros y bibliotecas deben contener todas las variables globales y funciones que se utilicen. En caso contrario no podrá generarse el ejecutable.

Por defecto el mandato `cc` llama `a.out` a este ejecutable, pero es posible ponerle un nombre mediante la opción `-o`. En realidad la última fase de composición del ejecutable no es realizada por `cc`, sino por el editor de enlaces `ld`. Este es otro mandato que se puede utilizar independientemente y que se encarga de la resolución de referencias externas y de montar el programa ejecutable. Así por ejemplo, la opción `-o` no es específica de `cc` sino de `ld`. El compilador invoca a `ld`, y le pasa esta opción.

-g > Mantiene información simbólica para que los depuradores de programas puedan actuar

-O > Optimiza el código ejecutable. Es incompatible con -g

-I directorio > Lugar donde el compilador buscará los documentos que aparecen en #include

Prácticas

1. Trabajemos sobre las diversas maneras de generar un ejecutable a partir de los ficheros de la siguiente práctica: `sucesion.c`, `fibonacci.c`, `ndnmas1.c`, `armonica.c`.
   (a.) Generar los objetos de cada uno de los ficheros por separado, y posteriormente genere un ejecutable con el nombre `sucesion`.
   (b.) Genere el ejecutable directamente pasándole a `cc` todos los fuentes implicados en el programa.
   (c.) Genere el ejecutable mezclando fuentes con objetos.

2. Se quiere realizar un programa que evalúe elementos de sucesiones de números. El programa mostrará en pantalla un menú para que el usuario elija entre tres opciones posibles:
   (a.) La sucesión de Fibonacci:

   $$x_{n+1}=x_n+x_{n-1}$$

   (b.) La sucesión n/n+1
   (c.) La sucesión armónica:

   $$\sum_{i=1}^{n}\frac{1}{i}$$

   El programa debe solicitar el número de elementos que se quieren visualizar, y una vez que termine de imprimirlos, volverá a mostrar el menú solicitando una nueva sucesión y número de elementos.

   Los ficheros `sucesión.c`, `fibonacci.c`, `ndnmas1.c` y `armonica.c` contienen parte del código de este programa. Añada todo lo que sea necesario para terminarlo y mejorarlo.

1.3 Bibliotecas

Desde un fichero C se llama a funciones que están implementadas en el propio fichero, o en otros ficheros C. También es posible llamar a funciones que están localizadas en alguna biblioteca, bien del sistema, bien creada por el propio usuario.

Una biblioteca es un fichero donde se "archivan" un conjunto de ficheros objetos. Desde un fichero C puede llamarse a una función que se encuentra localizada en una biblioteca. Es decir en un fichero objeto archivado en dicha bilioteca.

Evidentemente en el momento de compilar todo el programa para generar el ejecutable final, habrá que indicar también de qué biblioteca se utilizan funciones.

Tanto esto como el instante en el que se extrae de la biblioteca el código correspondiente, depende del tipo de biblioteca que se esté utilizando.

A. BIBLIOTECAS ESTÁTICAS

Estos son los tipos de bibliotecas más antiguos, y más extendidos tanto en Unix como en otros sistemas. Una biblioteca estática es el concepto tradicional de una biblioteca.

Si un programa llama a una función, el editor de enlace extrae su código de la biblioteca estática y lo añade al programa. Cualquier referencia a la función, se sustituye por la dirección de memoria que se le ha asignado a esta función. El resultado final de la compilación es por tanto un fichero ejecutable que incluye todo el código necesario para su ejecución.

Las bibliotecas se encuentran normalmente en el directorio `/lib` o en `/usr/lib`. En SVV4, se encuentran en `/usr/ccs/lib`. Los nombres de una biblioteca estática se forman de la siguiente manera:

`lib`*`nombre_bib`*`.a`

es decir, comienzan por la cadena `lib` seguida de un nombre, y deben terminar en `.a`.

Crear una biblioteca estática

El mandato para archivar y controlar varios ficheros objetos en un único fichero (la biblioteca) es `ar`. La forma de hacerlo es:

```
$ ar -r libnombre_bib.a [fich1.o [fich2.o] ...]
```

La opción `r` indica reemplazar los ficheros objetos que se pasan como argumentos por los ficheros del mismo nombre que se encuentran en la biblioteca. Es decir actualizará los objetos de la biblioteca que se indiquen. Esta misma opción se utiliza para crear la biblioteca por primera vez.

El mandato `ar` tiene además otras opciones para, por ejemplo, visualizar la lista de los ficheros objetos de la biblioteca, borrar un fichero objeto, extraerlo, etc. Esencialmente su sintaxis es:

```
$ ar opción [arg_opción] libnombre_bib.a [fich1.o ...]
```

donde *opción* además de -r puede ser:

`-d`	Elimina ficheros que se especifiquen.
`-q`	Añade ficheros a una biblioteca estática sin comprobaciones. No se comprueba si los ficheros ya existen en la biblioteca estática.
`-t`	Muestra una tabla de los ficheros de la biblioteca.
`-p`	Imprime los ficheros en la salida estándar.
`-x`	Extrae los ficheros de la biblioteca.

y donde *arg_opción* puede ser:

`-v`	Listado extenso de los ficheros.
`-c`	Suprime el mensaje de aviso que se produce al crear una biblioteca.

Prácticas

1. Liste los ficheros archivados en la biblioteca estándar `libc.a`
2. Cree una biblioteca llamada `suc.a` con los ficheros `fibonacci.o`, `ndnmas1.o` y `armonica.o`.
3. Extraiga la información de cada fichero en formato extenso (-tv).
4. Borre el fichero `fibonacci.o` y extráigalo de la biblioteca.

Utilizar una biblioteca estática

Cuando se quiere utilizar una biblioteca existen opciones disponibles para ello en el mandato `cc` (realmente son opciones para `ld`).

`-l`*`nombre_bib`*	esto le indica al editor de enlaces que en el programa se utilizan funciones que se encuentran en la biblioteca `lib`*`nombre_bib`*`.a`.
`-L`*`dir`*	El editor de enlaces busca en el directorio por defecto de las bibliotecas. Mediante esta opción las buscará en el directorio *`dir`*, antes que en el directorio por defecto.
-d*d*	En los sistemas en los que existan bibliotecas estáticas y dinámicas, el editor de enlaces posiblemente realizará por defecto un enlace dinámico. La opción -dn ó -dy permite realizar un enlace estático o dinámico, respectivamente.

Prácticas

Genere un ejecutable para el problema de las sucesiones a partir de la biblioteca estática anteriormente creada, de la siguiente manera:

```
cc  -L. -lsuc sucesion.c -o sucesion
```

B. BIBLIOTECA DINÁMICA

Una biblioteca dinámica (biblioteca de objetos compartidos "shared objects"), al igual que en las bibliotecas estáticas, contiene ficheros objeto. Sin embargo, el editor de enlace NO copia el código en el ejecutable, sino simplemente una referencia al nombre de la biblioteca junto con alguna información adicional.

En tiempo de ejecución se realiza esta operación, es decir, se extrae el código necesario de la biblioteca y se copia en memoria. Este código queda separado del código del fichero ejecutable de forma que permite que otros procesos que utilicen la misma biblioteca, utilicen la misma copia de dicho código.

Los nombres de las bibliotecas dinámicas empiezan con `lib` seguido de una cadena y terminan con `.so`:

`lib`*`nombre_bib`*`.so`

Ventajas de las bibliotecas dinámicas

- Se ahorra espacio puesto que el código extraído de una biblioteca dinámica no se copia en el ejecutable hasta el momento de ejecución.
- Ya que se comparte código en este tipo de biblioteca, disminuye la memoria dinámica necesaria para los procesos.
- Los ficheros ejecutables que utilicen bibliotecas dinámicas son más fáciles de mantener. Si se arregla un problema en una biblioteca dinámica no es necesario volver a recompilar los programas que la utilicen.

Desventajas de las bibliotecas dinámicas

- Si la biblioteca dinámica cambia de lugar, los ficheros ejecutables generados con anterioridad no encontrará dicha biblioteca. Es posible resolver este problema mediante variables especiales de la shell `LD_LIBRARY_PATH` y `LD_RUN_PATH`.
- Un programa generado con una biblioteca dinámica puede llegar a ocupar más espacio que uno generado con una biblioteca estática (lo que es bastante inusual). Esto podría ocurrir si la información adicional que se debe añadir en el ejecutable es mayor que el propio código a introducir.

Creación de una biblioteca dinámica

Al contrario que en las bibliotecas estáticas, el propio mandato cc se utiliza para crear este tipo de biblioteca. Volvamos a nuestro ejemplo de las sucesiones, la siguiente instrucción creará una biblioteca dinámica:

```
$ cc -K PIC -G -olibsuc.so fibonacci.c ndnmas1.c armonica.c
```

Las opciones tienen el siguiente significado:

-G	Indica al editor de enlace que cree una biblioteca dinámica.
-K PIC	Crea código independiente de la posición. Este código puede cargarse en cualquier sitio del espacio de direcciones del proceso. El programa se ejecutará sin modificar la página, esto mejora el sistema de manejo de memoria que subyace en el proceso de enlace dinámico.

Utilizar una biblioteca dinámica

La forma de indicar al editor de enlaces que se quiere utilizar una biblioteca dinámica es análoga al caso estático. Aquí simplemente habrá que indicar que la biblioteca es dinámica, lo que habitualmente es la acción por defecto.

```
$ cc -L. sucesion.c -lsuc
```

Prácticas

Compile con la biblioteca dinámica `libsuc.so`, y observe la diferencia de tamaño del programa ejecutable en comparación con que se genera utilizando una biblioteca estática.

1.4 La herramienta `lint`

Esta es una utilidad que intenta detectar posibles problemas en programas C. Es decir busca por construcciones y código que puede llevar a errores, que no sea portable o que no sea eficiente. Es decir la finalidad de `lint` no es la de encontrar errores sintácticos en el programa, ya que esto en sí es misión de `cc`. Su objetivo es encontrar errores escondidos, y mejorar el estilo del programa. También examina la concordancia entre tipos de datos de una forma más rigurosa que el compilador.

El mandato `lint` en particular comprueba:

- Sentencias inalcanzables
- Bucles infinitos y bucles en los que no se entra por el principio.
- Variables declaradas pero no utilizadas.
- Expresiones lógicas con valor constante.
- Funciones que retornan un valor en algunos lugares y no en otros.
- Llamadas a funciones con un número diferente de parámetros al de sus argumentos.
- Llamadas a funciones con tipos incoherentes al de sus argumentos.
- Funciones que no utilizan los parámetros que se le pasan.
- Sintaxis no portable o antigua.
- etc.

La sintaxis de esta orden es la siguiente:

```
$ lint [opciones] fich1 ...
```

`lint` funciona en dos pasadas al estilo de `cc`. Si se utiliza la opción `-c` genera un fichero con extensión `.ln` por cada uno de los ficheros fuentes C que se le pasen como parámetro. Estos ficheros son análogos a los `.o` generados por el compilador.

Es en la segunda pasada en la que `lint` comprobará la compatibilidad entre funciones de todos los ficheros que se le pasan como parámetro. Entre estos ficheros pueden aparecer ficheros terminados en `.ln` provenientes de `lints` anteriores.

Si no se pasa la opción `-c`, `lint` realizará las dos pasadas de forma automática, lógicamente sin generar ningún fichero intermedio `.ln`.

Algunas de las opciones disponibles son las siguientes:

`-a`	Suprime información sobre asignaciones de variables `long` a variables que no lo son.
`-b`	Suprime información sobre sentencias `break` que no son alcanzadas.
`-v`	Suprime información sobre argumentos no utilizados en funciones.
`-c`	Corresponde a la primera pasada de `lint`, se generan ficheros `.ln` que pueden ser utilizados posteriormente.
`-o` *bib*	`lint` creará una biblioteca de ficheros `.ln`, con el nombre `llib-l`*bib*`.ln`. Esta biblioteca puede ser utilizada posteriormente en una llamada a `lint` mediante la opción `-l`.
`-l` *bib*	Incluye la biblioteca de lint denominada `llib-l`*bib*`.ln`.

Prácticas

1. Pase `lint` al fichero `fibonacci.c`.
2. Genere un fichero `.ln` para el fichero `fibonacci.c`.
3. Pase `lint` a los ficheros C del problema de las sucesiones.
4. Igual que el caso anterior pero sustituya `fibonacci.c` por el fichero `fibonacci.ln`.

CAPÍTULO 8

COMPILACIÓN DE PROYECTOS

1.1 Proyectos software

Normalmente una aplicación software realiza una tarea de cierta complejidad. Por ello el enfoque racional utilizado para desarrollarla, consiste en dividir el problema global en varios subproblemas. Evidentemente es más fácil concentrar el esfuerzo en solucionar un punto concreto que intentar abordar toda la aplicación de una vez.

La filosofía de partir una aplicación en módulos y en submódulos claramente interrelacionados conlleva normalmente la creación lógica de un árbol de directorios, donde se sitúan los ficheros fuentes y cabecera (habitualmente también objetos, ejecutables, bibliotecas y otros ficheros para generación y depuración de los programas) que componen cada uno de dichos módulos.

Cuando el tamaño de una aplicación no es muy grande bastaría con meterlo todo en un mismo directorio. Pero cuando la aplicación es grande, compuesta por un gran número de ficheros, es inviable el situar todo esto en un mismo lugar, ya que el manejo de fuentes y objetos resulta demasiado aparatoso.

Por otra parte, las acciones implicadas en el desarrollo, pruebas y mantenimiento de un proyecto software son varias y muy repetitivas. Se hace imprescindible en muchas ocasiones la utilización de herramientas que facilitan todo el ciclo de vida de un desarrollo software. El entorno del sistema operativo Unix es muy adecuado para dicho ciclo, y dispone de varias utilidades con esta finalidad.

En el capítulo anterior ya vimos algunas órdenes básicas para la compilación de programas C. En este capítulo veremos una herramienta para hacer esta tarea de compilación más automática, y más fácil de manejar, sobre todo en el caso de no estar trabajando con pequeños programas de "juguete".

1.2 La utilidad `make`

La orden `make` proporciona un mecanismo para mantener versiones de programas actualizadas. Normalmente entre los ficheros que componen un programa existen muchas relaciones. Sin una herramienta autómatica como ésta, el programador tendría que recordar muchos detalles sobre cuáles eran estas dependencias, además de repetir una y otra vez una misma operación probablemente precedida de pequeñas operaciones de cambio y listado de directorio, compilaciones parciales, etc., además de tener que introducir largos argumentos.

Habitualmente `make` se utiliza para el desarrollo de software, pero su funcionamiento puede ser aplicado a otras tareas. Puede utilizarse en aquellas situaciones en las que existan dependencias entre ficheros, por ejemplo se usa con frecuencia en proyectos de documentación.

Para realizar sus operaciones automáticas, `make` se ayuda de un fichero creado y controlado por el programador, donde se sitúan las dependencias y acciones que se deben controlar. Este fichero puede denominarse de alguna de las siguientes maneras:

- `makefile`
- `Makefile`
- `s.[mM]akefile` (se tratarán como ficheros SCCS).

Por defecto la orden `make` busca en este orden estos ficheros en el directorio actual.

Si se utiliza alguno de ellos, normalmente la simple invocación de la orden `make` será suficiente para activar las acciones adecuadas dependiendo de los ficheros que se hayan modificado desde la última ejecución de `make`. En cualquier caso siempre existe la posibilidad de llamar al fichero de dependencias como se desee, y posteriormente indicárselo en la línea de mandatos a `make` mediante la opción -f.

En este fichero de dependencias existe lo que se llama el "target", el objetivo. Que es aquello que depende de una serie de ficheros. El objetivo puede ser o no un fichero.

Básicamente la acción realizada por `make` consiste en encontrar el objetivo que se debe construir, y comprobar que todos los ficheros de los cuales depende, existen y están actualizados. Es decir la fecha de todos los ficheros de los que depende el objetivo debe ser anterior a la fecha del objetivo (lógicamente si el objetivo no es un fichero, nunca estarán actualizadas las dependencias).

Si alguna dependencia no está actualizada el objetivo será construido (es decir, se realizarán las acciones especificadas para este objetivo en el fichero `makefile`). La utilidad `make` sólo regenerará aquello que no esté actualizado, dejando tal cual lo que esté al día.

Evidentemente esto puede ahorrar mucho tiempo en la compilación de grandes proyectos. En un proyecto software la modificación del más mínimo detalle requiere la generación de un nuevo ejecutable, pero muchas de las partes ya compiladas pueden reutilizarse sin necesidad de recompilarlas de nuevo.

1.3 El fichero `makefile`

Contiene las dependencias expresadas de la siguiente forma:

```
objetivo: [dependencias]
          acción1
          acción2
          .......
          acciónN
```

donde *objetivo* es aquello que se va a actualizar, y en *dependencias* se listan todos los ficheros de los cuales depende dicho objetivo. Las acciones que siguen en las siguientes líneas son las órdenes que se quieren ejecutar cuando las dependencias no estén actualizadas. Por defecto las acciones que se ejecuten se visualizarán en pantalla, salvo que se comiencen con un carácter "@". Habitualmente el formato del fichero `makefile` exige que cada acción comience por al menos un tabulador.

Al igual que en los programas de shell, cualquier secuencia de caracteres que empiece por el carácter "#" y termine con un salto de línea, se entiende como un comentario.

Si una línea de dependencia es demasiado larga, se puede continuar en la siguiente mediante el carácter "\".

1.4 Un ejemplo

El siguiente es un ejemplo que utilizaremos en éste y en posteriores capítulos. El programa realiza algunas transformaciones sobre su entrada estándar. El objeto del mismo es deformar el contenido del fichero que se le pase por la entrada estándar. Esta acción es ejecutada mediante la opción 0 del programa, y mediante otra opción (1), se realizará la transformación inversa.

Tendríamos por tanto un programa de juguete para cifrar algunos de nuestros ficheros de texto. El programa ha sido desarrollado con el objetivo de ilustrar herramientas para la implementación y depuración de aplicaciones, por lo que algo del código construido podría haberse optimizado bastante y organizado de otra manera.

Los ficheros fuentes que componen el programa son los siguientes:

- `cifrar.c`
- `menu.c`
- `leer.c`
- `inverticar.c`
- `invertitlinea.c`
- `imprimir.c`

Un estudio más a fondo del código será analizado en el próximo capítulo, en este sólo nos interesan los ficheros.

A. LAS DEPENDENCIAS

Un fichero `makefile` que nos permitiría compilar automáticamente el programa hasta obtener el ejecutable podría ser el siguiente:

```
cifrar:          cifrar.o menu.o leer.o invertircar.o \
                 invertirlinea.o imprimir.o
                 cc cifrar.o menu.o leer.o invertircar.o \
                 invertirlinea.o imprimir.o -o cifrar
cifrar.o:        cifrar.c cifrar.h
                 cc -c cifrar.c
menu.o:          menu.c cifrar.h
                 cc -c menu.c
leer.o:          leer.c cifrar.h
                 cc -c leer.c
inverticar.o:    invertircar.c cifrar.h
                 cc -c invertircar.c
invertirlinea.o: invertirlinea.c cifrar.h
                 cc -c invertirlinea.c
imprimir.o:      imprimir.c cifrar.h
                 cc -c imprimir.c
```

Como puede observarse los ficheros objetos (.o) dependen de sus respectivos fuentes C, y de los ficheros cabecera (.h) que incluyan. Esto quiere decir que si alguno de ellos (.c ó .h) se modificaran, `make` actualizaría el objeto correspondiente ejecutando la acción de compilación que se especifica en su dependencia.

1.5 Invocar a make

La sintaxis de la orden `make` es la siguiente:

```
$ make [-f fichmake ] [opciones] objetivo1 ...
```

donde algunas de las opciones posibles son las siguientes:

`f fichmake`	Indica a `make` que utilice este fichero de dependencias.
`i`	Indica a `make` que continúe aunque algunos mandatos se ejecuten con errores (en caso de no usarse esta opción se detendrá cuando se produzca un error).
`s`	No mostrar los mandatos que se están ejecutando (silencio).
`n`	Muestra los mandatos pero no los ejecuta.
`t`	Sólo actualiza la fecha de los objetivos, pero no los genera.

Los objetivos que se quieren actualizar se especificarán en la línea de mandatos. Expliquemos algunos ejemplos:

Compilación de un objetivo

```
$ make menu.o
```

Esta orden buscaría la línea de dependencia para el objetivo `menu.o` en el fichero de `make` que se encuentre en el directorio. En este caso los ficheros de los que depende son `menu.c` y `cifrar.h`. Si alguno de estos dos ficheros hubiera cambiado, entonces `make` ejecutaría la orden que se especifica a continuación de la relación de dependencia:

```
$ cc -c menu.c
```

Con ello se lograría generar el fichero objeto `menu.o`.

Prácticas

Ejecute la siguiente orden:

```
$ touch menu.c
```

y actualice el objetivo `menu.o` mediante la orden:

```
$ make menu.o
```

Compilación de múltiples objetivos

La siguiente orden haría la misma que la anterior pero actuaría sobre múltiples objetivos:

```
$ make leer.o imprimir.o
```

Prácticas

Actualice las fechas de los ficheros `leer.c` e `imprimir.c` y ejecute la orden anterior.

Generación del ejecutable

En el siguiente ejemplo, supongamos que está todo actualizado, pero que encontramos un error en el módulo `invertircar.c`. Una vez corregido nos gustaría probar si el programa actúa como pensamos. Tenemos por tanto que generar el ejecutable (`cifrar`) y probarlo. Para ello basta con hacer la orden:

```
$ make crifrar
```

Los pasos seguidos por `make` son los siguientes:

1. Descubre que el objetivo `cifrar` depende de una serie de ficheros .o, y que cada uno de éstos a su vez depende de un fichero .c.

2. Si algún fichero .o es más antiguo que su respectivo fichero fuente .c, entonces ejecutará la acción especificada. En nuestro ejemplo, ya que `invertircar.c` ha sido modificado, `make` generará el fichero `invertircar.o`.

3. Dado que algún fichero de los que depende `cifrar` se ha modificado, `make` generará el objetivo `cifrar`.

Si se ejecuta `make` cuando el objetivo está actualizado, simplemente informará de ello. Si se ejecuta la orden `make` sin pasarle ningún argumento, buscará el primer objetivo en el fichero de `make` y lo actualizará. Por lo tanto en nuestro ejemplo, la orden

```
$ make cifrar
```

es equivalente a

```
$ make
```

Prácticas

Realice la práctica descrita anteriormente. Una vez que todo esté actualizado vuelva a invocar a `make` y observe el mensaje.

Objetivos ficticios

En algunas ocasiones interesa realizar una acción siempre. En estos casos se suele definir un objetivo ficticio, es decir un objetivo que no es un fichero. Como el objetivo por tanto nunca se construye, nunca estará actualizado y por lo tanto siempre se realizará la acción a él asociada.

Un objetivo típico que se suele definir es aquél que desencadene la comprobación y generación en el caso necesario de todos los objetivos del fichero. Suponga, por ejemplo, que el fichero de `make` sirviera para generar varios ejecutables independientes, una forma de hacer que se examinaran todos ellos sería construir una dependencia mediante un objetivo ficticio:

```
todo:      ejecu1 ejecu2 ejecu3

ejecu1:    ........

ejecu2:    ........

ejecu3:    ........
```

1.6 Macros

La utilidad `make` puede definir variables de forma análoga a como se hace en la shell. A una macro puede asignársele una cadena de caracteres:

```
MACRO=cadena
```

La utilización de macros es cómoda cuando se quiere describir una lista larga de ficheros (quizás situados en diferentes directorios), y que se debe repetir más de una vez. O por ejemplo para describir opciones de compilación que se utilizarán en varias acciones, con lo cual, además de ser fácil de leer el fichero de `make`, será más fácil añadir o quitar opciones.

La forma de acceder al valor de una macro varía ligeramente de la utilizada en la shell:

```
$(MACRO)
```

Las variables de entorno de la shell son válidas dentro del fichero `makefile`, pero con notación de `make`. Además existen algunas macros predefinidas que pueden ser visualizadas mediante:

```
make -p 2>&1 | more
```

1.7 Combinándolo todo

Un fichero `makefile` completo para nuestro ejemplo puede ser:

```
FUENTES = cifrar.c menu.c leer.c invertircar.c \
          invertirlinea.c imprimir.c
OBJETOS = cifrar.o menu.o leer.o invertircar.o \
          invertirlinea.o imprimir.o

BIB=
LINT=lint
CFLAGS=-g
LP=/usr/bin/lp -d laser
cifrar:          $(OBJETOS)
                 $(CC) $(OBJETOS) $(BIB) -o cifrar
                 @echo "Generado \"cifrar\""
$(OBJETOS):      cifrar.h
borraro:
                 rm $(OBJETOS)
lint:            $(FUENTES)
                 $(LINT) $(FUENTES)
cifrar.o:        cifrar.c cifrar.h
                 cc $(CFLAGS) -c cifrar.c
menu.o:          menu.c cifrar.h
                 cc $(CFLAGS) -c menu.c
leer.o:          leer.c cifrar.h
                 cc $(CFLAGS) -c leer.c
invertircar.o:   invertircar.c cifrar.h
                 cc $(CFLAGS) -c invertircar.c
invertirlinea.o: invertirlinea.c cifrar.h
                 cc $(CFLAGS) -c invertirlinea.c
imprimir.o:      imprimir.c cifrar.h
                 cc $(CFLAGS) -c imprimir.c
```

```
    # El siguiente objetivo imprime los fuentes no
    # actualizados respecto al fichero "print".
print:  $(FUENTES)
        pr $? | $(LP)
        touch print
```

Comentarios

- Existen varias macros especiales, en el ejemplo anterior se ha utilizado $? cuyo valor es la cadena formada por los nombres de los ficheros más recientes que el objetivo.
- La utilidad `make` es inteligente en el sentido de que supone muchas dependencias implícitas, por ejemplo sabe construir ficheros objetos a partir de ficheros fuentes C. Son las llamadas reglas implícitas. En el caso anterior podríamos por tanto haber suprimido las relaciones entre objetos y fuentes.

CAPÍTULO 9

HERRAMIENTA DE DESARROLLO (CSCOPE)

(no existe en AIX)

1.1 Incrementar la productividad del programador

El sistema Unix dispone de numerosas herramientas para el desarrollo y la depuración de programas en C. Cuando lo que se quiere depurar es un pequeño programa de juguete, compuesto por uno, dos o tres módulos, no es necesario acudir a algunas de estas herramientas. Pero lo habitual es que las aplicaciones C estén formadas por un gran número de fuentes, probablemente distribuidos de una forma lógica, entre varios directorios.

Ya hemos visto en el capítulo anterior cómo automatizar la compilación de un proyecto mediante la utilidad `make`. Herramientas como ésta y la que veremos en el presente capítulo, aumentan considerablemente la productividad del programador.

Hasta hace poco los programadores tenían que moverse una y otra vez a lo largo de la estructura de directorios de un proyecto, buscando la definición de una función, la aparición de una variable, las llamadas a una función, etc. La orden `cscope` facilita esta tarea presentándo al usuario opciones, por ejemplo, para localizar cualquier cadena en múltiples ficheros, editar aquél fichero en el que se sospecha puede estar el error, sustituir una cadena por otra, comprobar los parámetros de llamada a una función, etc.

En una frase, `cscope` realiza acciones similares a las de `grep` y `vi` conjuntamente, de una forma cómoda, interactiva e inteligente.

1.2 Invocar cscope

La orden `cscope` construye una tabla de referencias cruzadas con todos los ficheros fuentes C (.c), ficheros cabecera (.h), ficheros `lex` (.l), y ficheros `yacc` (.y). Las utilidades `lex` y `yacc` son dos herramientas utilizadas para la generación automática de programas que realicen un análisis léxico y sintáctico, respectivamente. Es decir cuando la entrada al programa tiene un determinado formato o estructura sintáctica que debe cumplir.

Esta tabla de referencias cruzadas es un fichero que por defecto se denomina `cscope.out`, y es utilizado por `cscope` para encontrar la ubicación de variables, funciones, macros, etc. Siempre que se utilice `cscope` comprobará si esta tabla de referencias está actualizada, es decir si se ha modificado algún fichero o si la invocación se está realizando con la misma lista de ficheros que la última vez.

Por defecto se examinará en el directorio actual. Por lo que las siguientes instrucciones son equivalentes:

```
$ cscope
$ cscope *.[chly]
```

También es posible pasar una lista de ficheros para que sean examinados:

```
$ cscope conductor.c prueba1.c prueba2.c cab.h
```

La lista de ficheros puede crearse en un fichero, manualmente o utilizando algún mandato:

```
$ find . -name "*.[chly]" -print > listafich
```

y posteriormente pasarla a `cscope` mediante la opción `-i`:

```
$ cscope -i listafich
```

Prácticas

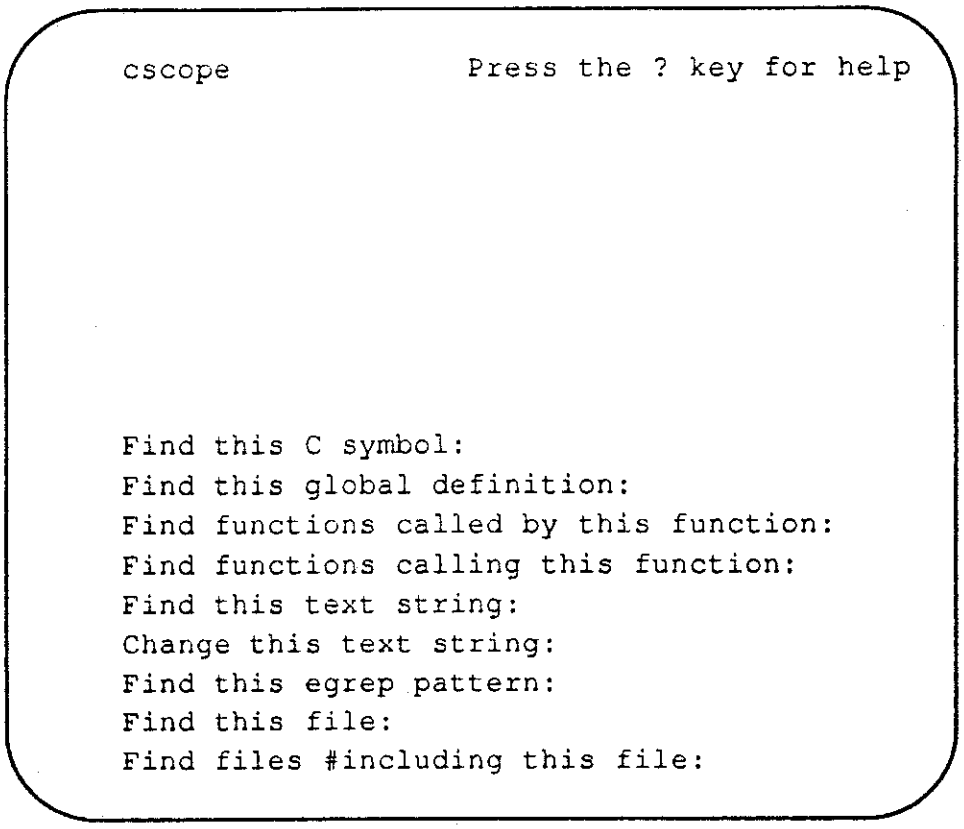
¿Cómo se podrían realizar los dos pasos anteriores en uno sólo, evitando la construcción del fichero intermedio?

A. MENÚ DE CSCOPE

Dependiendo del sistema las opciones de `cscope` pueden variar, en la versión SVV4 son las siguientes:

1. Encontrar un símbolo C.
2. Encontrar una definición global.
3. Encontrar las funciones que llama una función.
4. Encontrar las funciones que llaman a una función.
5. Encontrar una cadena de caracteres.
6. Sustituir una cadena de caracteres por otra.
7. Encontrar un patrón `egrep`.
8. Editar un fichero.
9. Encontrar aquellos ficheros que incluyen un determinado fichero cabecera.

Pantalla Inicial



Algunas de las teclas para actuar en este pantalla inicial son las siguientes:

Retorno ó tab ó ^m ó ↓	Una línea hacia abajo.
^p ó ↑	Una línea hacia arriba.
^r	Reconstruir la tabla de referencias cruzadas.
!	Abrir una shell interactiva.
^l	Redibujar la pantalla.
?	Lista de mandatos.
^d	Salir de `cscope`.

1.3 ¿Dónde busca cscope?

`cscope` busca los ficheros que se le indique. Existe una forma de que examine otros directorios. El mecanismo consiste en inicializar la variable VPATH con la lista ordenada de directorios separados por ":" donde se quiere buscar. Por ejemplo:

```
$ VPATH=.:./fuentes:/home/proyecto/fuentes/general
```

Busca los ficheros cabecera en los siguientes directorios y en ese orden:

1. El directorio actual.
2. Los directorios especificados por la opción -I (al estilo de `cc`).

   ```
   $ cscope -Icabs -I../cabs *.c fuentes/*.c
   ```

3. Los directorios estándares para los ficheros cabecera (normalmente `/usr/include`).

1.4 Prácticas

Una tarea que aparece con cierta frecuencia ante un programador es la de analizar qué es lo que hace un programa. Cuáles son las funciones que se utilizan en cada uno de los módulos, cómo se llaman entre sí. Seguir la ejecución del programa visualmente, etc. Esta tarea no suele ser nada fácil, por lo que `cscope` es de vital importancia.

Con la ayuda de `cscope` investigue los siguientes puntos sobre el problema de cifrado que ya se utilizó en el capítulo anterior.

1. Analice cuáles son las funciones que se utilizan en cada uno de los ficheros.
2. Dibuje un árbol de las llamadas entre funciones.
3. Descubra cuál es la transformación que se produce sobre los datos de entrada.
4. Descubra qué ficheros incluyen cada fichero cabecera usado en el programa.
5. Defina una macro para el número máximo de caracteres que puede contener cada línea, e insértela donde corresponda. Esto es cómodo hacerlo mediante la opción de sustituir una cadena por otra.
6. El programa escribe dos saltos de línea en vez de uno, mediante una inspección visual trate de determinar dónde está el error. Una invocación adecuada al programa puede ser la siguiente:

```
$ cifrar 0 < cifrar.c | more
```

El cifrado y descifrado de un mismo fichero debería dar un fichero idéntico:

```
$ cifrar 0 < cifrar.c | cifrar 1 > rescatado
$ cmp cifrar.c rescatado
$
```

CAPÍTULO 10

DEPURACIÓN DE PROGRAMAS

1.1 Corrección de un programa

Una vez que un programa está libre de errores de compilación estamos seguros de que no hemos introducido ninguna ilegalidad contra la sintaxis del lenguaje que hayamos utilizado. Es decir nuestro programa es sintácticamente correcto. Como resultado de nuestro esfuerzo obtenemos un fichero ejecutable, es decir, un fichero que se ejecutará simplemente con su invocación.

Esto en modo alguno nos asegura la corrección del programa, es decir, que realice exactamente lo que queremos. La experiencia nos dice que una de las fases que más tiempo consume en el ciclo de vida de un producto software es la de verificación y pruebas.

La simple inspección visual del programa, tal y como se ha hecho en el capítulo anterior, ayuda a localizar muchos errores, en algunos casos incluso más rápidamente que mediante algún otro mecanismo.

Es por tanto aconsejable realizar una comprobación visual pausada de todos los fuentes, realizando una ejecución mental del programa y cuestionando puntos como, la conherencia de tipos, los argumentos de llamada a una función, los puntos de salida de un bucle, etc. Posiblemente esta tarea ayudará a optimizar algunas partes del código.

Sin embargo, en muchas ocasiones el error está tan escondido que sólo mediante una herramienta de depuración podrá localizarse. La filosofía de un depurador consiste en ofrecer al usuario un control exhaustivo de la ejecución del programa. Permitiéndole acciones como:

- Poner puntos de parada del programa.
- Posibilidad de visualizar el contenido de variables locales y globales.
- Ejecución "paso a paso" del programa.
- Visualización de las funciones y argumentos que se encuentran anidadas en la pila.
- Visualización del código fuente.
- Ejecución del programa con diferentes argumentos de entrada.
- Modificar el contenido de variables.
- etc.

1.2 sdb: un depurador simbólico

La utilidad que Unix ofrece para realizar la depuración de programas C se denomina `sdb` ("simbolic **debugger**"). El calificativo de simbólico es debido a que el usuario puede referenciar las variables y las sentencias del programa donde aparecen, mediante sus nombres simbólicos y los números de las líneas en las que se encuentran en el código fuente C.

Todas las acciones antes comentadas sobre las posibilidades de un depurador son proporcionadas por `sdb`. Además puede usarse para examinar imagenes "core" de programas que han sido abortados. Es decir cuando un programa realiza una acción ilegal, el sistema Unix le envía una señal (veáse señales en el capítulo 13) para que termine. Esta situación ocasiona la creación de un fichero denominado `core`, que contiene una imagen de la memoria actual del proceso interrumpido. A esto se le suele denominar depuración "postmortem", en contraste con la depuración tradicional que permite controlar interactivamente la ejecución del programa.

1.3 La orden sdb

Para utilizar `sdb` con todas sus posibilidades es necesario compilar mediante la opción `-g` de `cc`. Esta opción le indica al compilador que almacene información en el ejecutable relativa a las variables y funciones del programa. Si no se utiliza esta opción todavía se puede depurar el programa, pero no existirán las relaciones con los nombres simbólicos del fichero fuente.

No se deben utilizar las opciones `-g` y `-O` (optimización del código generado) simultáneamente, a menos que el usuario conozca profundamente cómo se realiza la optimización.

```
cc -g fuente.c
```

Prácticas

Compile mediante la opción `-g` todos los fuentes C del programa de cifrado introducido en el capítulo anterior. Para ello basta con asignar esta opción a la macro CFLAGS de su fichero `makefile`.

La sintaxis de la orden `sdb` esencialmente es la siguiente:

```
sdb [ficheje[fichcore[listdirs]]]
```

donde:

ficheje	El programa que se va a depurar.
fichcore	El nombre de la imagen "core" (por defecto el fichero se denomina `core`) del proceso abortado. Si en el directorio actual existiera un fichero `core` y no quisiéramos usarlo, hay que escribir un "-" en este campo.
listdirs	Lista de directorios separados por ":" donde se encuentran los fuentes. Por defecto el directorio actual.

1.4 Órdenes sdb

Una vez que se invoca sdb, existen varias órdenes para la depuración de un programa. Usualmente el indicador que muestra esta herramienta es un "*" situado a la izquierda de la pantalla. La orden q abandonará la sesión sdb.

A. ÓRDENES DE EDICIÓN

sdb define el concepto de fichero actual y de línea actual referente a dicho fichero actual. En las órdenes donde se especifique lo contrario se entenderá por defecto el fichero y/o la línea actual.

Cuando sdb se utiliza con un fichero core, el fichero y la línea actuales indican el lugar donde se ha producido el fallo del programa (todas las relaciones con el fuente sólo aparecerán si el programa se ha compilado con la opción -g). Si se invoca sin fichero core, el fichero actual es el que contiene la función main(), y la línea actual es la primera de main().

p	Visualizar la línea actual.
l	Visualizar la línea correspondiente a la instrucción actual cuando examina un fichero core.
e [*fich*|*función*]	Cambiar el fichero actual por *fich*, o por el fichero que contiene la definición de *función*.
w	Visualizar una ventana de 10 líneas alrededor de la actual.
z	Visualizar la línea actual y las siguientes 9 líneas.
! *mandato*	Abrir una shell y ejecutar *mandato*. Por tanto podría editarse cualquier fichero fuente mediante vi.

B. ÓRDENES DE EJECUCIÓN

Algunas órdenes para controlar la ejecución del programa son las siguientes:

`r` *`args`*	Ejecutar el programa con los argumentos que se le pasen, si no se le dan argumentos se toman los de la última ejecución. Para ejecutar el programa sin argumentos se usa la orden R.
`s` *`num`*	Ejecutar *num* sentencias del programa (por defecto 1).
`c`	Continuar la ejecución de un proceso parado.
`línea``g`	Continuar la ejecución en la línea *`línea`* del fichero actual.
`función``:`*`línea`*`b`	Poner un punto de parada en la línea *`línea`* del fichero donde se encuentra *`función`*.
`B`	Listar todos los puntos de parada existentes.
`función``:`*`línea`*`d`	Borrar un punto de parada.
`D`	Borrar todos los puntos de parada existentes.
`k`	Terminar el programa.

S num

C. ÓRDENES PARA OBTENER INFORMACIÓN

`t` Visualizar una traza de la pila, es decir la secuencia completa de las llamadas a funciones que están anidadas en ese momento.

`T` Mostrar la línea superior de la pila.

`[fun:]variable/clm` Mostrar el contenido de esta `variable` según la longitud `l` y el formato `m` (por defecto se utiliza el definido en el fichero fuente para esa variable). `c` indica el número de bloques de tamaño `l` que se quieren visualizar. El valor de `l` puede ser:

- `b` un byte
- `h` dos bytes
- `l` cuatro bytes

y el valor de `m`:

- `c` carácter
- `d` decimal
- `u` decimal sin signo
- `o` octal
- `x` hexadecimal
- `f` punto flotante de precisión simple
- `g` punto flotante de precisión doble
- `s` cadena de caracteres
- `p` puntero a función

`variable!valor` Asignar `valor` a `variable`.

D. ÓRDENES PARA LOCALIZAR CADENAS

/expr_regular/	Buscar la siguiente cadena a partir de la línea actual que cumple la expresión regular (al estilo de ed) especificada. El último carácter / puede omitirse, salvo cuando le siga el operador de punto de parada.
?expr_regular?	Realiza la misma operación que la anterior expresión, salvo que busca a partir de la línea actual hacia atrás.

1.5 Prácticas

1. Ejecute el programa de cifrado que ha compilado con la opción `-g` de la siguiente forma:

   ```
   $ cifrar 0 < cifrar.c
   Memory fault(coredump)
   $
   ```

   Este programa tiene un error que provocará una salida atípica del mismo, y la generación de un fichero `core`.

2. Invoque a `sdb` utilizando depuración "postmortem" de la siguiente forma:

   ```
   $ sdb cifrar
   31:                                   buftemp[j]=lin[i];
   *
   ```

3. Utilice las órdenes de edición para localizar dónde aparece exactamente esta sentencia.

4. Visualice cuáles son las funciones anidadas en ese instante.

5. Visualice el valor de las variables `i` y `j`.

6. Visualice el contenido de los arrays `buftemp` y `lin`.

7. Salga de `sdb` y ejecútelo de nuevo, pero ahora ignorando el fichero `core`.

   ```
   $ sdb cifrar -
   ```

8. Ponga un punto de parada antes de llamar a la función donde se encuentra la sentencia en la que finalizó el programa.

9. Ejecute ahora el programa mediante la orden:

   ```
   *r 0 < cifrar.c
   ```

10. Si el programa no ha sido abortado antes de llegar a este punto, continúe la ejecución paso a paso, y utilice todas las facilidades de `sdb` para resolver el problema.

CAPÍTULO 11
CONTROL DE VERSIONES

1.1 Versiones de un producto

Normalmente un producto software pasa por varias etapas. Desde las primeras especificaciones hasta el resultado final de una aplicación software, seguramente se habrán introducido muchas modificiones y ampliaciones. Este resultado final probablemente sólo constituya una versión del producto, a la que seguirán otras, conforme se observen errores, posibles mejoras y adaptación a otros equipos y entornos.

Todavía podemos hacer un nuevo refinamiento si descendemos al nivel de los ficheros fuente que componen cada versión de un mismo producto. Cada uno de estos ficheros sufrirá con el tiempo diferentes modificaciones y transformaciones.

Detectamos tres problemas diferentes que deben ser resueltos:

- ¿Qué ocurre si queremos volver a alguna versión anterior de la aplicación software, o de algún fichero fuente?. Esta situación puede ocurrir por varias razones: una marcha atrás en las especificaciones, aparición de errores en un programa fuente que obligan a retroceder hasta algún punto anterior, se requiere comparar una antigua con una nueva versión, etc.
- Toda aplicación software de tamaño grande o mediano está compuesta de varios módulos fuente. Es por tanto necesario llevar un control detallado de qué versión de cada uno de estos ficheros compone una versión particular del producto.
- Lógicamente, ser capaz de recuperar versiones anteriores también es un requisito indespensable. Esto nos proporcionaría además un método de salvaguardia de nuestros ficheros. En más de una ocasión, la eliminación por descuido de algún fuente podrá ser subsanable si se dispone de una herramienta de control de versiones.

1.2 El sistema de control de código fuente (SCCS)

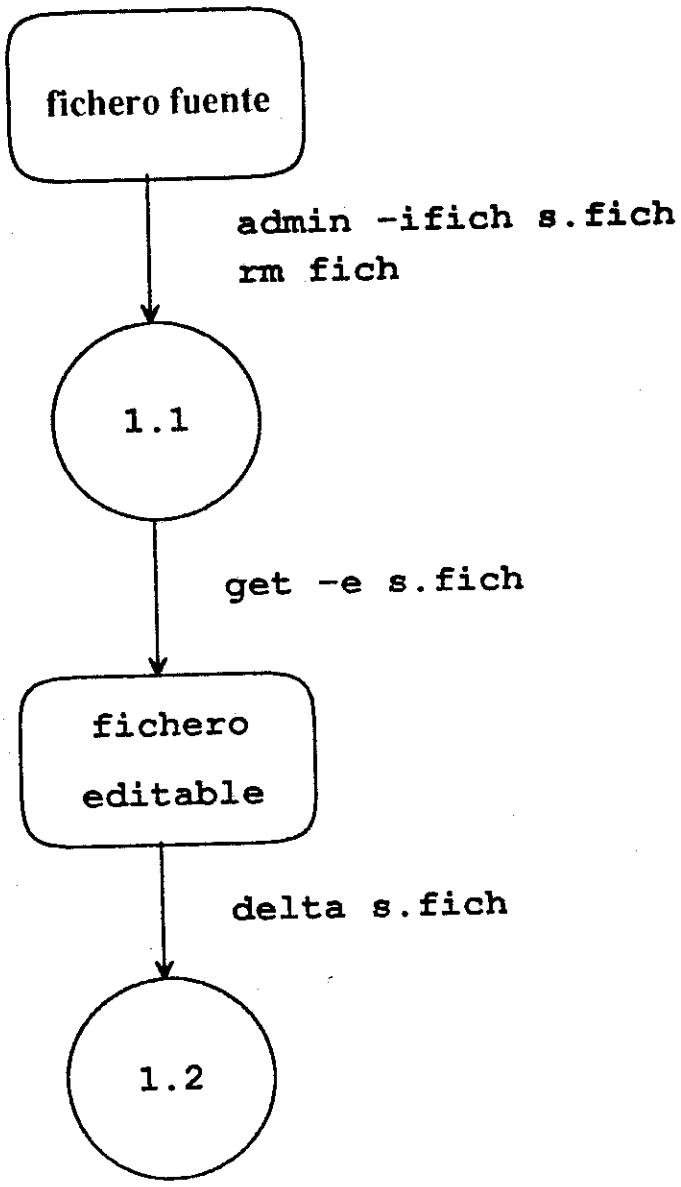
Una herramienta disponible en Unix para ayudar en la tarea de controlar las versiones de un fichero fuente es SCCS (Source Code Control System). Originalmente fue diseñada para la gestión de software, pero es capaz de gestionar cualquier fichero de texto. Por ello SCCS puede utilizarse también para controlar documentos.

SCCS controla todas las versiones de un fuente, almacenando toda la información necesaria para ello en un fichero (denominado el s-fichero). Se pueden añadir nuevas modificaciones (deltas), para construir nuevas versiones.

Las versiones se numeran empezando en 1.1, y en cualquier momento se puede solicitar a SCCS que recupere alguna versión pasada.

1.3 Fases en el control de un fichero

Utilizar lo esencial de SCCS es muy fácil. Veamos a continuación los pasos implicados en la gestión de un fichero.



A. CREAR UN FICHERO SCCS

La primera tarea que se debe realizar es la de poner el fichero bajo control de SCCS. Para ello se dispone del mandato `admin`. La finalidad de este mandato no es exclusivamente la de crear un fichero SCCS, sino que también se puede utilizar para administrarlo (de ahí su nombre). Con la orden:

```
$ admin -ifichero s.fichero
No id keywords (cm7)
```

se crea el s-fichero asociado con el fichero que se quiere controlar. Este fichero debe proporcionarse mediante la opción -i, y el s-fichero debe obligatoriamente empezar por el prefijo "`s.`".

El mensaje que reponde este primer mandato no es un error sino sólo un aviso. Sólo indica que no existen palabras claves en el fichero. Puede obtenerse una explicación más detallada de este mensaje mediante:

```
$ help cm7
```

Dado que el fichero está ahora bajo control de SCCS el fichero fuente puede borrarse:

```
$ rm fichero
```

Prácticas

Ponga el fichero `amayu.c` bajo el control de SCCS.

B. RECUPERAR UN FICHERO

El mandato SCCS que permite recuperar alguna versión de un fichero es `get`. Mediante esta orden es posible obtener cualquier versión de un fichero (la última por defecto), tanto para utilizarla, como para generar otra versión a partir de ella.

Por ejemplo para recuperar la última versión del fichero `amayu.c` ejecute lo siguiente:

```
$ get s.amayu.c
1.1
36 lines
No id keywords (cm7)
```

Observe que esta orden ha extraído la última versión del fichero fuente (la única existente hasta el momento). Al utilizar `get` sin opciones, se entiende que lo que se quiere es obtener la última versión del fichero, para cosultarla o usarla de alguna manera, pero no para generar una versión a partir de ella. Es decir la versión recuperada no es modificable, de hecho se le asigna permiso sólo de lectura.

No es posible generar una nueva versión a partir del fichero extraído, ni siquiera asignando permiso de escritura (SCCS no está preparado para recibir una nueva versión). Para generarla se debe utilizar la opción `-e`.

```
$ get -e s.amayu.c
1.1
new delta 1.2
36 lines
```

SCCS construye un fichero, ahora sí con permiso de escritura, con la última versión de `amayu.c` (1.1). Además crea el llamado p-fichero (`p.amayu.c` en nuestro caso) que contiene información necesaria para cuando posteriormente se genere una nueva versión mediante el mandato `delta`.

C. GUARDAR UNA NUEVA VERSIÓN

Una vez que se dispone de un fichero con permiso de escritura, se deben hacer los cambios que se vean convenientes mediante cualquier editor. Una vez que estos cambios hayan sido probados y validados (en el caso de un programa lógicamente habría que compilarlo y probarlo), se debe guardar esta nueva versión del fichero.

El mandato para hacerlo es `delta`.

```
$ delta s.fichero
comments?
```

El mandato espera un s-fichero como argumento, y responde solicitando un comentario sobre las características de esta nueva versión.

El mandato `delta` apunta las diferencias utilizando el mandato `diff` de UNIX. Los cambios realizados se graban en el fichero `s.`*`fichero`*. Tanto el fichero fuente como el p-fichero desaparecen una vez que `delta` se ejecuta correctamente.

Prácticas

Modifique el fichero `amayu.c` para que cumpla la norma ANSI de definición de funciones, posteriormente guárdelo como una nueva versión:

```
$ delta s.amayu.c
comments? Version ANSI de amayu.c
No id keywords (cm7)
1.2
2 inserted
2 deleted
34 unchanged
```

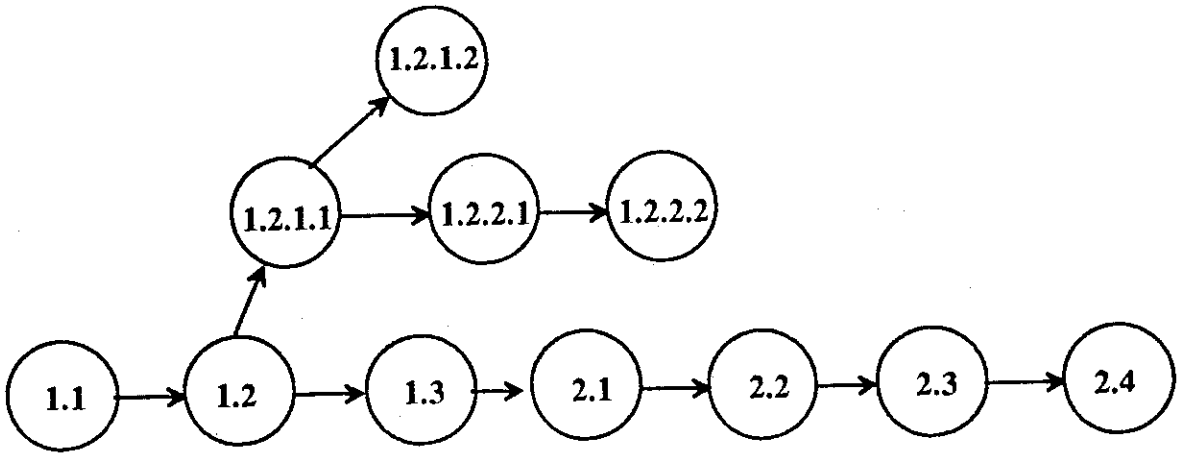
1.4 Numeración de versiones

Cada versión de un fichero es conocida como un delta. Puede pensarse en un delta como un nodo de un árbol que tiene como raíz la primera versión del fichero (1.1). El número que se le asocia a cada versión es denominado SID (Identificación SCCS). Este número está compuesto normalmente por dos números (1.1, 1.2, 2.1, ...) a los que se les llama "versión.nivel".

Cuando se genera un nuevo delta se incrementa usualmente el número de nivel. Sólo cuando la diferencia entre una versión y la siguiente es significativa el usuario puede decidir incrementar el número de versión.

Sin embargo, en algunas ocasiones surge la necesidad de crear una nueva rama en el árbol de deltas. Suponga por ejemplo que se está trabajando en la versión 2.4, y que se necesita actualizar la versión 1.2. Es decir un usuario de esta versión 1.2 ha detectado un problema, que requiere sea arreglado de forma inmediata. SCCS ofrece la posibilidad de generar la versión 1.2.1.1. El SID de esta versión tiene ahora dos números más: "versión.nivel.rama.secuencia".

Un ejemplo del árbol de deltas de un fichero se esquematiza en la siguiente figura:



1.5 Opciones de get

Ya hemos visto cómo `get` puede usarse para obtener la última versión de un fichero bajo control de SCCS. Veamos ahora cómo puede utilizarse para obtener una versión concreta, o para abrir una nueva rama en el árbol de deltas.

Obtener una versión

Para ello basta con utilizar la opción `-r`. Por ejemplo para obtener la versión 2.2 del esquema anterior:

```
$ get -r2.2 s.fichero
```

O por ejemplo la 1.2.2.1

```
$ get -r1.2.2.1 s.fichero
```

Si se omite el número de nivel, se recuperará la versión con el número de nivel más alto con el número de versión especificado. Así, por ejemplo:

```
$ get -r1 s.fichero
```

recuperaría el delta 1.3 en el esquema anterior.

Incrementar el número de versión

Tal y como comentamos anteriormente, cuando los cambios introducidos son importantes es adecuado incrementar el número de versión. Por ejemplo en el esquema, en vez de continuar con el delta 1.4, se salta al 2.1. Esto puede hacerse de la siguiente forma:

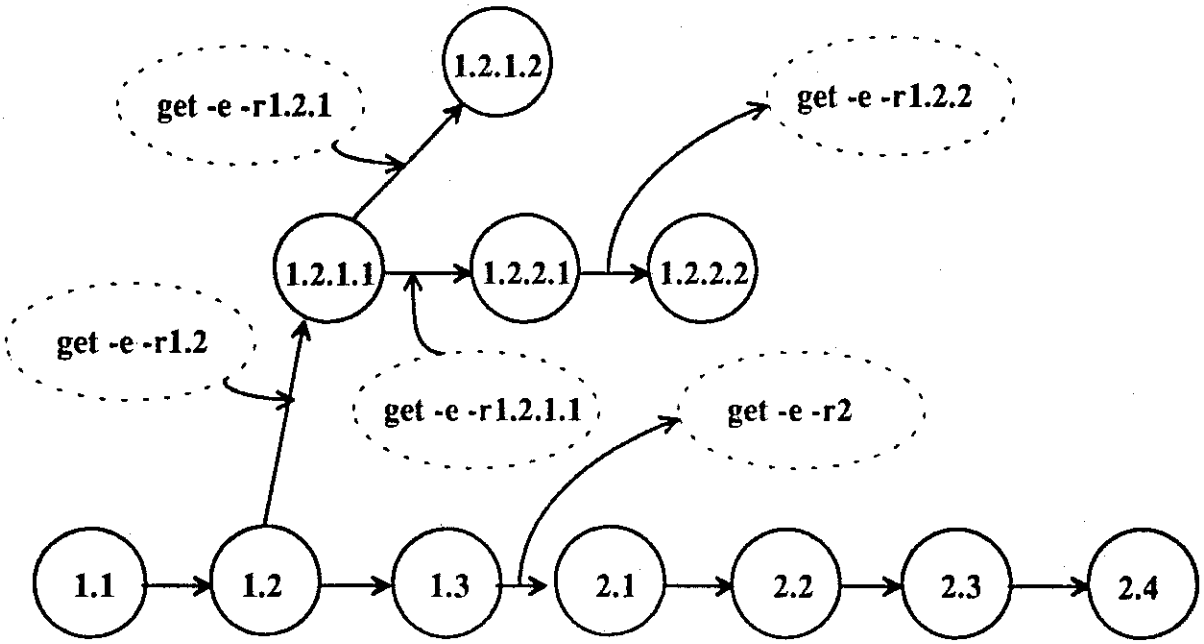
```
$ get -e -r2 s.fichero
```

Crear una nueva rama

Planteemos el caso del esquema, se quiere crear la rama que empiece en 1.2.1.1. Para ello bastará con utilizar la orden `get` de la siguiente forma:

```
$ get -e -r1.3 s.fichero
```

En la siguiente figura se recogen las llamadas a `get` que se deben realizar para construir el árbol de deltas del esquema.



En la información sobre `get(1)` se describe con todo detalle los SIDs que se pueden obtener, dependiendo del estado actual del árbol de deltas. En cualquier caso es aconsejable no complicarlo demasiado.

Prácticas

Pruebe a generar el árbol de versiones del esquema.

1.6 Palabras claves

SCCS ofrece un mecanismo mediante el cual es posible incluir en un fichero recuperado (no para edición), información que identifique a ese fichero. El sistema consiste en que el usuario escriba algunas palabras claves en el fichero fuente. Cuando una versión concreta es extraída, SCCS sustituya las palabras claves que encuentre por información relativa a dicha versión.

La forma general de una palabra clave es `%X%`, donde `X` es alguna letra mayúscula. Algunas de estas palabras clave son:

%M%	Nombre del fichero.
%R%	Versión.
%L%	Nivel.
%B%	Rama.
%S%	Secuencia.
%I%	Número de SID completo.
%H%	Fecha actual.
%T%	Hora actual.

Prácticas

Obtenga una versión modificable del fichero fuente `amayu.c`. Inserte las palabras claves que considere oportuno en un comentario, y guarde esta nueva versión. Obténgala (mediante `get` sin la opción `-e`), y observe su contenido.

1.7 El mandato `what`

El mandato `what` no forma parte estrictamente de SCCS. La acción que realiza es la de buscar en el fichero que se le indique cualquier ocurrencia de la cadena @(#), e imprimir lo que esté a su derecha hasta encontrarse uno de los limitadores siguientes:

"	dobles comillas
\	barra invertida
\n	nueva línea
\0	nulo no imprimible
>	mayor que

Prácticas

Ejecute el siguiente mandato:

```
$ what /usr/bin/vi | more
```

Hay tres palabras clave de identificación asociadas a una cadena de formato `what`:

`%Z%`	se expande a los cuatro caracteres de la cadena `@(#)`
`%W%`	equivale a `%Z% %M% %I%`
`%A%`	equivale a `%Z% %Y% %M% %I% %Z%`

Prácticas

1. En el programa `amayu.c` inserte la definición del siguiente array global:

   ```
   char *idsccs="%W%";
   ```

2. Cree un nuevo delta, y obtenga una versión no editable del programa.

3. Compílelo, y ejecute `what` sobre el ejecutable.

4. Otra forma de hacer esto mismo es utilizar la directiva `#ident` del preprocesador de `cc`. La cadena que siga a esta directiva aparecerá en el ejecutable final. Interesa por tanto pasar a esta directiva una cadena que comience por @(#), para que sea detectada por `what`. Escriba la siguiente línea:

   ```
   #ident "%A%"
   ```

   en una versión editable del fichero `amayu.c`. Guárdela mediante `delta`, recupérela mediante `get`, compile y finalmente ejecute `what` sobre el ejecutable generado.

1.8 Otros mandatos

A continuación se listan algunos de los mandatos disponibles en SCCS:

Mandato unget

Si se extrae un fichero para modificarlo (`get -e`), y posteriormente se decide no crear una nueva versión (es decir guardarlo mediante `delta`), es posible deshacer esta operación dejando todo como estaba:

```
$ unget s.fich
```

Mandato sact

Este mandato devuelve información sobre un fichero que ha sido extraído para edición (`get -e`). En concreto informa sobre el delta actual, el delta que se va a crear, el usuario que obtuvo el fichero editable, y la fecha y la hora en que se obtuvo.

```
$ sact s.fich
```

Mandato rmdel

En SCCS existe un mandato para borrar el último delta (el más reciente) de una rama. Este delta tiene que existir, es decir no se puede intentar borrar un delta que todavía no ha sido guardado.

```
$ rmdel -rSID s.fich
```

Mandato prs

El mandato `prs` se utiliza para extrae información sobre los deltas bajo control de SCCS. Permite además formatear adecuadamente esta información. Su sintaxis es básicamente:

```
prs [opciones] s.fich
```

Por defecto `prs` informa sobre todos los deltas existentes para un fichero.

Las opciones disponibles son las siguientes

`r`*SID*	Indica un delta al cual referir la solicitud de información. Por defecto el último.
`e`	Da información sobre todos los deltas posteriores al SID especificado.
`l`	Da información sobre todos los deltas anteriores al SID especificado.
`d [`*esp_datos*`]`	En *esp_datos* se indica cuál es la información y el formato requerido. Puede contener: • Texto que se imprimirá sin modificaciones. • Un salto de línea (\n), ó un tabulador (\t). • Palabras clave, que se sustituyen por su valor correspondiente.

Una lista de algunas de las palabras clave reconocidas es (la lista completa puede consultarse en `prs(1)`:

Palabra clave	Sustitución	Valor	Formato
`:Dt:`	D-delta, R-delta borrado	D ó R	S
`:I:`	SID	:R:.:L:.:B:.:S:	S
`:F:`	nombre_fichero SCCS	texto	S
`:DL:`	Estadísticas	`:Li:/:Ld:/:Lu:`	S
`:Li:`	Líneas insertadas	nnnnn	S
`:Ld:`	Líneas borradas	nnnnn	S
`:Lu:`	Líneas sin cambios	nnnnn	S
`:R:`	Número de versión	nnnn	S
`:L:`	Número de nivel	nnnn	S
`:B:`	Número de rama	nnnn	S
`:S:`	Número secuencia	nnnn	S
`:D:`	Fecha de creación	AA/MM/DD	S
`:Dy:`	Año de creación	nn	S
`:Dm:`	Mes de creación	nn	S
`:Dd:`	Día de creación	nn	S
`:C:`	Comentarios	texto	S
`:Y:`	Tipo de módulo	texto	S
`:Q:`	Descripción	texto	S
`:M:`	Nombre de módulo	texto	S
`:BD:`	Cuerpo del fichero	texto	M
`:GB:`	Cuerpo obtenido	texto	M
`:MR:`	Números MR ("modification request")	texto	M

S significa línea simple y M línea múltiple (es decir se imprimirá un salto de línea después de la sustitución).

En el caso por defecto la información que se ofrece es:

```
":Dt:\t:DL:\nMRs:\n:MR:COMMENTS:\n:C:"
```

Mandato cdc

Mediante este mandato es posible cambiar el comentario asociado con un delta.

```
cdc  -rSID s.fich
```

Mandato sccsdiff

Muestra las diferencias entre dos deltas de un fichero usando para ello el mandato `diff`.

```
sccsdiff -rSID1 -rSID2 s.fich
```

Mandato comb

Este mandato se utiliza para reducir un fichero SCCS. El método consiste en generar un programa de shell mediante este mandato. Una vez que se tiene este programa, se usará para reducir el tamaño del fichero SCCS. También puede usarse para que sólo se tenga en cuenta la información a partir de un delta concreto, eliminando de esta manera los deltas anteriores.

```
comb s.fich > script_red
chmod u + x script_red
script_red s.fich
```

CAPÍTULO 12

TRATAMIENTO DE PROCESOS

1.1 Estructura de un proceso

Un programa es una colección de instrucciones y datos guardados en un fichero regular. Este fichero simplemente se marca como ejecutable mediante el mecanismo de permisos implementado en Unix.

Por otra parte, un proceso es un entorno de ejecución de un programa, compuesto por un segmento de instrucciones, un segmento de datos, además de la pila y del montón ("heap") de donde se extrae la memoria dinámica.

Cuando se ejecuta un programa se activa un proceso. Los segmentos de instrucciones y de datos del proceso se inicializan con las instrucciones y datos de dicho programa, respectivamente.

Un programa puede ejecutarse simultáneamente en diferentes procesos sin existir relación funcional alguna entre ellos. Sin embargo, en algunos sistemas ambos procesos podrían utilizar el mismo segmento de instrucciones por consideraciones de ahorro de memoria.

1.2 Atributos de un proceso

Un proceso tiene un gran número de atributos que lo caracterizan. A continuación vamos a enumerar brevemente algunos de ellos:

- Directorio de trabajo actual
- Directorio raíz

  Un proceso podría ver (aunque no es usual) un punto del árbol de directorio como el directorio raíz.

- ID de usuario y grupo real

  Un proceso tiene un identificador de usuario real y otro de grupo real, los cuales coinciden con los identificadores de usuario y de grupo, respectivamente, del usuario responsable de la creación del proceso.

- ID de usuario y grupo efectivo

  Normalmente coinciden con los identificadores reales respectivos. Son utilizados para determinar los permisos de acceso a ficheros. En caso de que el proceso ejecute un programa con el bit s de usuario y/o de grupo activo, entonces el identificador de usuario y/o de grupo efectivo respectivamente, será igual al usuario propietarios y/o grupo propietario del programa.

- Terminal controlador

  El terminal al que está asignado el proceso.

- Identificador de sesión

  Identificador de la sesión a la que pertenece un proceso. Una sesión tiene como mucho, un único controlador de terminal, y éste sólo puede tener una sesión asignada.

- Identificador de grupo de procesos.

  Cada proceso pertenece a un grupo de procesos, lo cuales tienen una relación lógica entre sí (este atributo se hereda de padres a hijos). La shell interactiva es "líder" de una sesión. Un proceso que no sea líder de un grupo de procesos puede crear un nuevo grupo. De esta forma se constituye en líder y rompe la conexión con el terminal controlador.

1.3 Interfaz de un programa

Al ejecutar un programa en Unix recibe dos colecciones de datos desde el proceso que lo invocó: los argumentos y el entorno.

El interfaz completo de un programa C es el siguiente:

```
extern char **environ;
main(int argc, char *argv[], char *envp[])
```

`argc` es el número de parámetros que se le pasan al programa en su invocación,

`argv` es un array de cadenas, donde cada uno de los elementos de este array corresponde con el parámetro i-ésimo (el 0 es el nombre del programa),

`envp` es un array de cadenas, donde cada una de ellas es de la forma:

`variable=valor`

Este array describe las variables del entorno del proceso y sus valores. La variable global `environ` también se puede utilizar para acceder al entorno del proceso.

Prácticas

Escriba un programa en C que visualice todas las variables del entorno y sus valores. Utilice tanto la variable del interfaz como la externa.

1.4 Modos de un proceso

Un proceso (un programa en ejecución) corre en dos modos diferentes: modo usuario y modo núcleo (kernel). Suelen existir dos pilas separadas, una cuando el proceso está ejecutándose en modo núcleo, y otra cuando lo hace en modo usuario.

Cuando un proceso necesita modificar datos del propio núcleo del sistema operativo (por ejemplo para crear un nuevo proceso), es imprescindible que lo haga bajo su control. Por esto existen puntos de entrada al núcleo denominados "llamadas al sistema".

A. LLAMADAS AL SISTEMA

La apariencia de una llamada al sistema es básicamente la misma que la de una función C. Y la forma de utilizarla no discierne de ella desde el punto de vista del programador.

Una llamada al sistema implica por tanto un cambio de contexto, es decir el proceso pasará de ejecutarse desde modo usuario a modo núcleo, y una vez terminada la acción regresará de nuevo al modo usuario. Normalmente este cambio de contexto consume más tiempo, por lo que es deseable utilizar las llamadas al sistema sólo cuando realmente sea necesario.

Una llamada al sistema devuelve un valor. Este valor será -1 si se ha producido un error en su ejecución, en caso contrario este valor será distinto de -1, y en la mayoría de los casos tiene una interpretación. Por ejemplo podría ser el número de bytes leídos o escritos, el descriptor asignado, etc.

Todas las acciones sobre procesos que vamos a realizar en este capítulo se activan mediante llamadas al sistema.

B. VARIABLE ERRNO

Existe una variable entera del sistema denominada `errno`, que contiene un código indicando el último error producido en una llamada al sistema. Por tanto se debe consultar justo después de comprobar que una llamada devolvió -1.

En algunos sistemas la información sobre los códigos de los errores y su explicación se puede obtener invocando al manual on-line de la siguiente forma:

```
$ man 2 intro
```

y en otros, consultando:

```
$ man 2 errno
```

La lista de estos mensajes se guarda en el array externo:

```
extern char *sys_errlist[];
```

y el número máximo de errores definidos se encuentran en la variable:

```
extern int sys_nerr;
```

En SVV4 existe el procedimiento `strerror()` que evita el acceso a `sys_errlist`:

```
strerror(errno);
```

Una función que visualiza el mensaje asociado al último error producido por una llamada al sistema es la siguiente:

```
void perror(const char *s)
```

Esta función escribirá en la salida estándar la cadena que se le pase como parámetro seguida de dos puntos (siempre y cuando no sea nula), y después informará sobre el mensaje de error correspondiente a `errno`.

Prácticas

Escriba un programa C para visualizar todos los mensajes de error del sistema.

1.5 Creación de un proceso

En Unix todos los procesos son creados mediante la llamada al sistema `fork()` (salvo algunos que se activan en el arranque de la máquina). La acción realizada por esta llamada es la de crear un nuevo proceso que es una copia exacta del proceso que la invocó. El nuevo proceso es conocido como el hijo, y el que lo ha creado como el proceso padre.

Cuando `fork` termina existen dos copias exactas de un mismo proceso, es decir los dos procesos continuarán su ejecución exactamente en el punto donde `fork` ha sido llamado. Mediante el valor retornado en la llamada a `fork` se puede discernir:

- Si se ha producido un error y no ha sido posible crear el nuevo proceso.
- ¿Quién es el hijo?
- ¿Quién es el padre?

El convenio acordado para averiguar esto es el siguiente:

- Lógicamente un valor de retorno de -1 indica error en la llamada al sistema.
- El proceso que obtiene un valor de retorno de 0 es el hijo.
- El proceso que obtiene un valor de retorno distinto de 0 es el padre. Además este valor es el identificador del proceso hijo.

El hijo hereda del padre prácticamente todos sus atributos, salvo algunas cuestiones como por ejemplo:

- Los identificadores de proceso y los identificadores del proceso padre de cada uno de ellos son lógicamente diferentes.
- Los dos procesos tienen los mismos ficheros abiertos y los mismos descriptores asignados, compartiendo además el mismo puntero al fichero. Esto quiere decir que si un proceso avanza o retrocede en dicho fichero, esta operación también afecta al otro. Sin embargo si un proceso cierra el fichero el otro todavía lo mantendrá abierto.
- Los tiempos de ejecución del proceso hijo se inicializan a 0.

Recuerde que el proceso padre será una copia del proceso hijo, y que por tanto justo después de la creación, los dos procesos tendrán las mismas variables con los mismos contenidos. Esta es una forma implícita de pasar información del proceso padre al hijo.

Prácticas

Analice y ejecute el siguiente programa:

```
#define ERROR   -1
#include <stdio.h>
main()
{
int     id;

     switch (id=fork())
     {
        case ERROR:
           perror("No se ha podido crear el proceso\n");
           break;
        case 0:
           printf("Soy el hijo, ID:%d\n", id=getpid());
           printf ("HIJO:Ahora puedo realizar alguna
                                          tarea\n");
           printf ("HIJO:independiente de mi proceso
                                          padre\n");
           break;
        default:
           printf("Soy el padre\n");
           printf("PADRE:He creado el proceso con
                                          ID:%d\n",id);
           break;
     }
}
```

1.6 Ejecutar un programa

A. LLAMADA AL SISTEMA EXEC

Después de que el núcleo del sistema Unix ha arrancado, la única forma de ejecutar programas es utilizar la llamada al sistema `exec`. Un proceso, como ya se comentó, es un programa en ejecución, un entorno de ejecución compuesto básicamente por un segmento de instrucciones, un segmento de datos, y un segmento dinámico. Cuando desde un proceso se invoca a la llamada al sistema `exec`, estos segmentos del proceso serán inicializados con los segmentos de instrucciones y de datos del programa que se invoca.

Mediante esta llamada al sistema es posible ejecutar programas ejecutables (con instrucciones máquina para el microprocesador) y programas de shell.

Existen diversos formatos para `exec` dependiendo de los argumentos del programa que se quiera ejecutar. Básicamente a `exec` se le pasa el fichero ejecutable y sus parámetros. Las variantes de esta llamada son las siguientes:

```
int execl(char *ruta, char *arg0,..., char *argn, (char *)0);

int execv(char *ruta, char *argv[]);

int execle(char *ruta,char*arg0,...,char *argn,(char)*0, char *envp[]);

int execve(char *ruta, char *argv[], char *envp[]);

int execlp(char *fich, char *arg0, ..., char *argn, (char *)0);

int execvp(char *fich, char *argv[]);
```

Las diferencias se encuentran en:

- Si los argumentos del programa que se quiere ejecutar se dan, en una lista explícita terminada en un puntero nulo, o mediante arrays al estilo del interfaz de la función `main`. Esto es importante ya que en tiempo de compilación puede que no se conozca el número de parámetros del programa.
- Si el programa que se invoca se busca en la lista de directorios de la variable PATH, o se debe dar la ruta absoluta o relativa del fichero ejecutable.
- Si se quiere pasar el entorno del proceso explícitamente mediante un puntero, o automáticamente mediante la variable `environ`.

No hay retorno una vez que se ha invocado a la llamada `exec`. Si vuelve es a causa de algún error, por lo que no hay necesidad de comprobar que el valor de retorno es -1.

El proceso cargado con el nuevo programa mantiene muchos de sus atributos, algunas de las cuestiones que varían son las siguientes:

- Se configurarán por defecto aquellas señales que estuvieran preparadas para ser capturadas (debido a que las instrucciones para manejar una señal han desaparecido).
- Si el programa tiene algún bit `s` asignado, al proceso se le concederán los permisos del propietario (usuario o grupo) de dicho programa.

Prácticas

1. Analice y ejecute el siguiente programa C. Después de ejecutarlo, invoque `ps -f`, y observe que `arg0` es utilizado simplemente como etiqueta del proceso:

```
#define ERROR   -1
#include <stdio.h>
main()
{
int     id;
char    *ptr0=NULL;

    switch (id=fork())
    {
        case ERROR:
            perror("No se ha podido crear el
                                    proceso\n");
            break;
        case 0:
            printf("Soy el hijo, ID:%d\n",
                                    id=getpid());
            execve("/usr/bin/sleep", "DORMIR", "30",
                                    (char *)0);
            perror("Error al ejecutar \"sleep\"\n");
            break;
        default:
            printf("PADRE:He creado el proceso con
                                    ID:%d\n",id);
            printf("PADRE:Adios\n");
            break;
    }
}
```

2. Modifique el programa anterior para que el proceso hijo no pueda ejecutar la llamada `execve`. Esto se puede conseguir por ejemplo dando una ruta incorrecta.

3. Modifique el programa anterior para que no sea necesario dar la ruta del mandato `sleep`.

B. SENTENCIA EXEC

También existe en este caso una sentencia de shell cuya finalidad es la misma que la de la llamada al sistema, pero en este caso puede ser utilizada desde la shell interactiva o desde un programa C. La sintaxis es la siguiente:

```
$ exec [ mandato ... ]
```

Prácticas

¿Qué ocurrirá al ejecutar el siguiente mandato?

```
$ exec date
```

C. ALTERNATIVA FORK/EXEC

Observe que las dos funciones anteriores actúan independientemente, pero a su vez están íntimamente ligadas. En la gran mayoría de los casos siempre se utiliza `fork` para crear un nuevo proceso, y posteriormente se utiliza `exec` para cargar dicho proceso con un programa.

En algunas ocasiones, sin embargo, puede ser interesante utilizar sólo una de ellas. En otros casos se necesita hacer alguna operación antes de invocar definitivamente a `exec`. Podrían haberse implementado las dos operaciones en una sola, pero de está manera se ofrece más flexibilidad.

1.7 Terminar un proceso

Análogamente a como se puede terminar explícitamente un programa de shell mediante la sentencia `exit`, también es posible hacerlo desde un programa C. La llamada al sistema que realiza esta acción es

```
exit(estado);
```

donde *estado* es un entero que indica cómo ha terminado el proceso. Por convenio, cero indica correcto, mientras que un número entre 1 y 255 indica que el proceso ha finalizado anormalmente. El proceso padre del proceso que termina puede obtener este valor. En el caso de que este proceso sea la shell, el valor es depositado en la variable $?.

Prácticas

Escriba un programa C que pida memoria al sistema mediante la función `malloc`. Si esta función no puede conseguir la memoria, entonces devuelve NULL. En este caso salga con `exit` y un valor de retorno no cero. En caso contrario el valor de retorno será cero.

Fuerce un error pasándole a `malloc` un tamaño de 0 bytes. Compruebe que el valor devuelto por `exit` se puede obtener leyendo la variable $?.

1.8 Esperar a un hijo

Un proceso puede esperar hasta que algunos de sus procesos hijos termine. Esto realmente es lo que hace la shell cuando ejecuta alguna orden en primer plano.

A. LLAMADA AL SISTEMA WAIT

Desde un programa en C es posible esperar a que un proceso hijo termine mediante la siguiente llamada al sistema:

```
int wait(int *estado)
```

Si el proceso que la invoca tiene procesos hijos, `wait` esperará hasta que uno de ellos termine. No hay forma de especificar con `wait` por qué hijo se quiere esperar.

En la variable `estado` se almacena la información de finalización del proceso hijo. En ella se codifica si el proceso hijo ha finalizado después de llamar a `exit` y el valor de retorno, o si ha terminado por la recepción de alguna señal.

La llamada `wait` devuelve como retorno -1 si se ha producido algún error, o en caso contrario el identificador del proceso hijo que ha terminado.

Analizar la información de estado

Para analizar este valor existen algunas macros:

`WIFEXITED(estado)`	Retornará un valor no cero si el proceso terminó normalmente (mediante `exit`).
`WEXITSTATUS(estado)`	Si la macro anterior devuelve un valor no cero, retornará el estado de salida (`exit`) que devolvió el proceso hijo.
`WIFSIGNALED(estado)`	Retornará un valor no cero si el proceso terminó por la recepción de una señal.
`WTERMSIG(estado)`	Si la macro anterior devuelve un valor no cero, retornará el número de la señal que obligó al proceso hijo a terminar.

¿Cuándo termina wait?

La llamada `wait` terminará en los siguientes casos:

- En cuanto termine algún proceso hijo.
- Si el proceso padre recibe una señal mientras está esperando, `wait` retornará con -1, y la variable `errno` tendrá el valor `EINTR` (código de error que indica que una llamada al sistema ha sido interrumpida por una señal).
- Si no hay hijos, `wait` retornará -1 y la variable `errno` contendrá el valor `ECHILD`.

Prácticas

1. Escriba un programa en C que realice la misma función que la sentencia de shell `wait`.

B. SENTENCIA WAIT

En la shell también existe una sentencia que permite a un proceso esperar por un proceso hijo, bien desde la shell interactiva, o bien desde un programa de shell. Esta sentencia utiliza la llamada anterior para finalmente esperar. Su formato es:

```
wait [idproceso]
```

donde *idproceso* es el identificador del proceso hijo en segundo plano por el cual se quiere esperar (si no se especifica número se esperará por todos, es decir hasta que todos hayan acabado).

Prácticas

1. Pruebe a esperar por un proceso hijo de la shell:

```
$ sleep 20 &
[1]        1428
$ wait 1428
```

2. Pruebe a esperar por varios procesos hijos simultáneamente. La sentencia `wait` no terminará hasta que todos los procesos terminen:

```
$ sleep 20 &
[1]        1429
$ sleep 40 &
[2]        1480
$ wait
```

3. Construya un programa de shell con la siguiente línea:

```
sleep 30 &
```

Ejecute este programa de shell, y antes de que termine, localice el identificador de este proceso en segundo plano (`sleep`) e intente esperar por él, ¿qué ocurre?

C. PROCESOS HUÉRFANOS Y ZOMBIS

Cuando un proceso hijo se queda sin padre, es decir su proceso padre muere, el proceso huérfano es adoptado automáticamente por el proceso `init`, el proceso con identificador 1.

Por otra parte, existe también una situación especial cuando un proceso hijo muere, y su proceso padre no está esperando por él. En este caso, el proceso que ha muerto descarga los segmentos que estuviese utilizando, pero mantiene sin embargo una entrada en la tabla de procesos.

No consume recursos, y la única información que conserva es la que tiene que entregar al proceso padre cuando finalmente espere por él. Una vez que esto ocurre, el proceso zombie desaparece para siempre. Estos procesos aparecen como <defunct> cuando se ejecuta el mandato `ps -e`. Existe una forma de evitar que esta situación se produzca, y consiste en configurar la señal SIGCHLD para que sea ignorada (véanse las señales más adelante).

```
signal(SIGCHLD, SIG_IGN)
```

Prácticas

1. Ejecute el programa `zombi` que se da a continuación en segundo plano, pero antes de que termine debe ejecutar la orden siguiente a fin de observar los procesos zombi existentes:

```
$ ps -e | grep defunct
$ cat zombi.c
#define ERROR   -1
#include <stdio.h>
main()
{
int     id;

        /*signal(SIGCHLD,SIG_IGN);*/ /* Esto si no se
quiere que el proceso quede zombie*/
     switch (id=fork())
     {
           case ERROR:
                 perror("No se ha podido crear el
                                        proceso\n");
                 break;
           case 0:
                 printf("HIJO:ID=%d. ADIOS\n",
                                        id=getpid());
                 break;
           default:
                 printf("PADRE:He creado el proceso con
                                        ID:%d\n",id);
                 sleep(20);
                 printf("PADRE:ADIOS\n");
                 break;
        }
}
```

CAPÍTULO 13

SEÑALIZACIÓN ENTRE PROCESOS

1.1 Comunicación entre procesos

Unix es un sistema multiproceso en el que es imprescindible una eficiente comunicación entre procesos.

En múltiples ocasiones surge la necesidad de que un proceso avise a otro, o que le comunique alguna información. Los mecanismos de comunicación han evolucionado y mejorado conforme han ido apareciendo nuevas versiones del sistema.

La mejora en la eficacia de los mecanismos ya existentes, y la incorporación de nuevas y más versátiles posibilidades están ocasionando que, cada vez sea mayor el número de grupos de desarrollo que utilizan estas características de comunicación inter-procesos. Lo que a su vez repercute en su perfeccionamiento y robustez, así como en la aparición de herramientas apropiadas para facilitar la implementación de aplicaciones bajo la óptica del multiproceso.

En los sistemas anteriores al Sistema III la comunicación se realizaba mediante ficheros, señales y cauces ("pipes"). En el Sistema V de AT&T y Xenix de Microsoft se añadieron semáforos, mensajes y memoria compartida.

En la Versión 4.0 del Sistema V se han mejorado estas tres nuevas posibilidades haciéndolas más rápidas y eficientes.

Los métodos de comunicación disponibles en Unix los podemos clasificar en cuatro grupos:

- Señales.
- Usar ficheros para sincronización entre procesos.
- Usar cauces y fifos.
- Inter-Process Communication (IPC) que engloba mensajes, semáforos y memoria compartida.

En este capítulo se pretender exponer exclusivamente cómo funcionan las señales Unix, ya que la completa exposición del resto de posibilidades exigiría por sí mismo todo un curso.

1.2 Señales

Las señales en Unix constituyen un mecanismo que permite avisar a un proceso (o a varios) de la ocurrencia de un suceso. Los tipos de eventos que puede desencadenar el envío de una señal son de diferente naturaleza:

- Un error producido en la ejecución de un proceso. El núcleo del sistema le enviará una señal.
- Un evento externo al proceso. Por ejemplo una señal de alarma, la terminación de un proceso hijo, fin de sesión, el usuario ha pulsado la tecla de interrupción, etc.

Dependiendo de la señal, el proceso que la recibe puede adoptar diferentes comportamientos:

- Dejar que el proceso se comporte según la acción definida por defecto para dicha señal. Las acciones por defecto variarán según el tipo de señal, en particular éstas pueden ser:
  i. Terminar, ante lo cual el proceso que recibe la señal finalizará su ejecución.
  ii. Terminar y generar un fichero `core` en el directorio actual. Este fichero es una imagen de memoria del proceso en el momento de abortar. Como se explicó en el capítulo 10, `sdb` utiliza este fichero para analizar el motivo de la anormal finalización del proceso.
  iii. Parar temporalmente, lo cual dormirá el proceso hasta que no se active posteriormente mediante la señal correspondiente, o finalmente se aborte.
- Ignorar la señal.
- Ejecutar una función asignada por el proceso a la señal recibida. Una vez que esta función finaliza el proceso continuará por donde fue interrumpido.

1.3 Tipos de señales

Cada una de las señales definidas en el sistema tiene un código, información que puede lograrse ejecutando el mandato:

```
$ kill -l
 1) HUP          12) SYS         23) STOP
 2) INT          13) PIPE        24) TSTP
 3) QUIT         14) ALRM        25) CONT
 4) ILL          15) TERM        26) TTIN
 5) TRAP         16) USR1        27) TTOU
 6) IOT          17) USR2        28) VTALRM
 7) EMT          18) CHLD        29) PROF
 8) FPE          19) PWR         30) XCPU
 9) KILL         20) WINCH       31) XFSZ
10) BUS          21) URG
11) SEGV         22) POLL
```

De una versión Unix a otra, los códigos y los tipos de señales difieren ligeramente. En el fichero `<signal.h>` o en `<sys/signal.h>` del directorio `/usr/include` se encuentra normalmente la lista de señales disponibles. Por ejemplo las señales en Unix SVV4 son las siguientes (versión a la que nos referiremos habitualmente):

Nombre	código	Explicación	Acción por defecto
SIGHUP	1	Fin de sesión.	Terminar
SIGINT	2	Interrupción (kill).	Terminar
SIGQUIT	3	Interrupción (quit).	core
SIGILL	4	Instrucción ilegal.	core
SIGTRAP	5	Señal de traza o punto de parada.	core
SIGABRT	6	Abortar (instrucción IOT).	core
SIGEMT	7	Emulación (instrucción EMT).	core
SIGFPE	8	Excepción de punto flotante.	core
SIGKILL	9	Terminación	Terminar
SIGBUS	10	Error de bus.	core
SIGSEGV	11	Violación de memoria (paginación/segmentación).	core
SIGSYS	12	Argumento incorrecto en la llamada al sistema.	core
SIGPIPE	13	Escritura en un cauce en el que nadie está leyendo.	Terminar
SIGALRM	14	Alarma de reloj.	Terminar
SIGTERM	15	Terminación software.	Terminar
SIGUSR1	16	Señal 1 para usuarios.	Terminar
SIGUSR2	17	Señal 2 para usuarios.	Terminar
SIGCHLD	18	Terminación de un hijo.	Ignorar
SIGPWR	19	Caída de alimentación/rearranque.	Ignorar
SIGWINCH	20	Cambio del tamaño de ventana.	Ignorar
SIGURG	21	Información urgente en el socket.	Ignorar
SIGPOLL	22	Ha ocurrido un evento seleccionable.	Terminar
SIGSTOP	23	Parada temporal (señal).	Parar
SIGTSTP	24	Parada temporal de usuario pedida desde un terminal.	Parar
SIGCONT	25	Continuación del proceso.	Ignorar
SIGTTIN	26	Intento de lectura en el terminal en segundo plano.	Parar
SIGTTOU	27	Intento de escritura en el terminal en segundo plano.	Parar
SIGVTALRM	28	Vencido el temporizador virtual.	Terminar
SIGPROF	29	Vencido el temporizador de planificación.	Terminar
SIGXCPU	30	Consumido el tiempo máximo de CPU.	core
SIGXFSZ	31	Excedido el límite de tamaño del fichero.	core

1.4 Enviar señales

Sin considerar los procesos cuyo identificador de usuario efectivo es el superusuario, un proceso sólo puede enviar señales con éxito a aquellos procesos que tengan el mismo identificador de usuario real, o el mismo identificador de usuario efectivo que el proceso que envía la señal. Existen varias formas de enviar una señal.

A. SECUENCIA DE TECLAS

Desde el terminal es posible enviar algunas señales al proceso que se esté ejecutando en primer plano. Lo habitual es querer abortar o parar temporalmente la ejecución de alguna orden. Para realizar esta acción habitualmente están disponibles las siguientes secuencias de teclas:

`<Ctrl-C>`	Envía la señal 2, `SIGINT`. Esta operación causa por defecto la finalización del proceso.
`<Ctrl-\>`	Envía la señal 3, `SIGQUIT`, además de provocar por defecto la finalización del proceso, se generará un fichero `core`.
`<Ctrl-Z>`	Envía la señal 24, `SIGTSTP`. El proceso que reciba esta señal parará temporalmente.

Estas secuencias son configurables mediante la orden `stty`.

Prácticas

Averigüe cuál es la secuencia de caracteres que envía la señal `quit`, (use `stty -a`). Invoque el siguiente mandato:

```
$ sleep 20
```

y antes de que termine pulse esa secuencia de caracteres. Observe la creación del fichero `core`.

B. MANDATO KILL

Desde la shell se puede enviar cualquier señal mediante el mandato `kill`. Su formato es el siguiente:

```
kill [ -número_señal ] iden_proceso
```

Este mandato enviará la señal *número_señal* al proceso con identificador *iden_proceso*. En caso de no especificarse señal se enviará la número 15 (SIGTERM).

La forma correcta de matar un proceso es enviar la señal 15. Si un proceso no se deja matar se puede utilizar la señal 9 (SIGKILL). Esto sólo debe emplearse como último recurso, ya que esta señal no se puede capturar (ni ignorar).

Prácticas

Escriba un programa de shell que muestre en formato `select` los procesos del usuario indicado (no opción indica el propio usuario, y la opción -a indica todos los usuarios) y que envíe la señal especificada (mediante opción), a los procesos seleccionados. Ejemplo:

```
enviar -15
enviar -9 -u ingusr1
enviar -3 -a
```

C. LLAMADA AL SISTEMA KILL

Desde un programa C también es posible enviar una señal a un proceso. Esto se consigue mediante la llamada al sistema `kill`. Su formato es el siguiente:

```
int kill( int pid, int señal);
```

donde `señal` indica el número de señal que se envía. La llamada devuelve 0 si se ejecuta correctamente y -1 si se ha producido algún error. El valor de la variable `pid` tiene varias interpretaciones:

pid	Interpretación
> 0	Identificador de proceso al que se quiere enviar la señal.
= 0	envío de la señal a todos los procesos (salvo el 0 y el 1) del mismo grupo de procesos que el emisor.
= -1	envío de la señal a todos los procesos (salvo el 0 y el 1) con propietario real igual al propietario efectivo del proceso que envía.
< -1	envío de la señal a todos los procesos del mismo grupo de procesos que el proceso con identificador |pid|

1.5 Capturar señales

Ya se comentó anteriormente cuáles eran los comportamientos que un proceso puede adoptar ante la llegada de una señal. La configuración para que tome una acción u otra puede llevarse a cabo desde dos lugares distintos, como frecuentemente: desde un programa de shell (mediante la sentencia `trap`), o mediante la llamada al sistema `signal`.

A. LA SENTENCIA TRAP

Desde la shell se puede "atrapar" una señal mediante el mandato `trap`. Su formato es el sigueinte:

```
trap [ mandato ] [ [señal] ...]
```

donde

mandato es el mandato que se va a ejecutar en el momento de recibir alguna de las señales que se indican posteriormente.

señal es el número o el nombre simbólico de una señal que se quiere configurar.

Con estos argumentos se puede configurar la señal para realizar cualquiera de las acciones, según se indica en la siguiente tabla:

Uso de trap	Explicación	Ejemplo	Acción
`trap - s ...` `trap s ...`	Resetear las señales a su configuración por defecto.	`trap - 1 2` `trap 1 2`	Las señales 1 y 2 se resetean a la acción por defecto (terminar)
`trap "" s ...`	Ignorar las señales.	`trap "" 1 2 15`	Ignorar las señales 1, 2 y 15
`trap m s ...`	Se ejecutará el mandato *m* al recibir las señales.	`trap "rm tmp.$$;exit 2" 2`	Cuando se reciba la señal 2, se ejecutará ese mandato.

No tiene efecto cualquier intento de capturar una señal, que al entrar en la shell actual estaba configurada para ser ignorada .

El mandato `trap` sin argumentos mostrará las señales ignoradas y capturadas en ese instante y en el proceso que lo invoca.

B. LA LLAMADA AL SISTEMA SIGNAL

Para configurar adecuadamente las señales desde un programa C se utiliza la llamada al sistema `signal`. Su formato es:

```
void (*signal (int sig, void (*disp)(int))) (int)
```

La explicación de esta definición es la siguiente:

`signal`	es una función con dos argumentos: `sig` y `disp`.
`sig`	es el número de la señal que se quiere configurar, y
`disp`	es la "disposición" de la señal; desde el punto de vista de C, "*un puntero a una función con un argumento de tipo entero*" (el argumento indicará la señal recibida). La función no devuelve ningún valor (void). A *`disp`* pueden asignársele esencialmente tres valores: SIG_DFL, SIG_IGN y una función implementada en el programa.

La función devuelve un puntero a una función del mismo tipo que su segundo argumento:

```
void (*)(int)
```

Esto es así puesto que la función `signal` devuelve `SIG_ERR` si se ha producido un error, y devuelve la última configuración definida para la señal en caso de que se haya ejecutado correctamente.

Señales especiales

Las señales SIGKILL y SIGSTOP no pueden ser ignoradas ni capturadas. Esto quiere decir que si se pasan como argumento a `signal`, esta llamada devolverá un error.

A continuación se presenta en una tabla la forma de llamar a `signal` según la acción que se quiera realizar:

Uso de signal	Explicación	Ejemplo
`signal(`*`señal`*`, SIG_DFL)`	Resetear *señal* a su configuración por defecto.	`signal(SIGHUP,SIG_DFL)`
`signal(`*`señal`*`, SIG_IGN)`	Ignorar *señal*.	`signal(SIGINT,SIG_IGN)`
`signal(`*`señal`*`, `*`función`*`)`	Al recibir *señal* se ejecutará *función*.	`signal(SIGINT,manejar)`

C. DIFERENCIAS DE SIGNAL CON BSD

Entre la versión BSD y SYSTEM V existen diferentes primitivas para tratar con señales. Tanto la llamada al sistema `kill` como `signal` existen en ambas versiones. Pero, mientras que la primera se comporta igualmente en las dos versiones, la segunda presenta varias diferencias en su funcionamiento. Veamos en qué difiere `signal` en SYSTEM V respecto a BSD:

- Las señales pueden enmascararse (como veremos posteriormente), lo que impide que una señal determinada pueda interrumpir la ejecución de una función que maneje la recepción de una señal anterior. En el caso BSD una vez que se está tratando una señal, ésta se añade automáticamente a la lista de señales enmascaradas. En el caso SYSTEM V no, lo cual significa que la misma señal puede volver a interrumpir mientras se esté tratando una anterior.
- Salvo para las señales SIGILL, SIGTRAP y SIGPWR, en SYSTEM V, antes de ejecutarse la función manejadora de una señal, dicha señal es configurada a su acción por defecto (SIG_DFL). Esto implica que en esta función lo primero que debe hacerse es reconfigurar la señal mediante `signal`, si se quiere que sea dicha función la que permanentemente trate la señal.
- En SYSTEM V, si un proceso recibe una señal mientras está ejecutando una llamada al sistema, ésta será interrumpida. Dicha llamada retornará el valor -1, y la variable `errno` contendrá el valor EINTR.

Prácticas

1. El siguiente programa captura la señal SIGINT (^C).

```
$ cat autosignal.c
#include <stdio.h>
#include <signal.h>
#define         ERROR           -1
void manejar(int numero);
main()
{
        signal(SIGINT, manejar);
        if (kill(getpid(),SIGINT)==ERROR)
        {
                perror("autosignal");
                exit(2);
        }
        printf("Programa termina\n");
        exit(0);
}
void manejar(int numero)
{
          signal(SIGINT, manejar);
          printf("Principio de manejador de se#al %d\n",
numero);
          sleep(5);
          printf("Final de manejador de se#al\n");
}
```

Analice y ejecute el programa de la siguiente forma:

```
$ autosignal
Principio de manejador de se#al 2
```

¿Suponiendo que el programa se ejecuta en SYSTEM V, qué ocurre si en esto momento se teclea uno o más ^C?

2. Elimine la llamada a `signal` que se realiza dentro de la función `manejar` (encerrándola entre comentarios). ¿Cuál debe ser ahora el comportamiento del programa?.

D. LA LLAMADA AL SISTEMA SIGSET

La llamada al sistema `sigset` sólo existe en SYSTEM V. Se utiliza igual que `signal` pero su comportamiento es semejante a la señal `signal` en BSD. Es decir, se enmascará una señal antes de ejecutar la función que la trata. La señal se desenmascará al terminar la función, dejándola en el mismo estado en el que se encontraba antes de producirse. Por lo tanto no se debe volver a configurar la señal dentro de la función, con lo cual la asignación es permanente.

También esta llamada al sistema dispone de una disposición más: SIG_HOLD. La acción que se realiza es la de añadir la señal que se indique a la lista de señales enmascaradas, dejando sin cambios la última disposición que se hubiera realizado sobre dicha señal (SIG_DFL, SIG_IGN o función).

Prácticas

Analice cómo se ejecutará ahora el programa anterior, utilizando `sigset` en vez de `signal`.

```
$ cat autosignal.c
#include <stdio.h>
#include <signal.h>
#define         ERROR           -1
void manejar(int numero);
main()
{
        sigset(SIGINT, manejar);
        if (kill(getpid(),SIGINT)==ERROR)
        {
                perror("autosignal");
                exit(2);
        }
        printf("Programa termina\n");
}
void manejar(int numero)
{
          printf("Principio de manejador de se#al %d\n",
numero);
          sleep(5);
          printf("Final de manejador de se#al\n");
}
```

E. OTRAS LLAMADAS AL SISTEMA

Llamada sighold

La llamada `sighold` añade la señal que se indique a la lista de señales enmascaradas.

```
int sighold (int señal)
```

Por tanto

```
sighold(señal)
```

es equivalente a

```
sigset(señal, SIG_HOLD);
```

Llamada sigrelse

La llamada `sigrelse` elimina la señal indicada de la lista de señales enmascaradas.

```
int sigrelse(int señal)
```

Llamada sigpause

La llamada `sigpause` elimina la señal indicada de la lista de señales enmascaradas, de forma análoga a la anterior, pero además bloquea al proceso hasta que reciba una señal.

```
int sigpause(int señal)
```

Llamada sigignore

La llamada `sigignore` configura una señal para que se ignore.

```
int sigignore(int señal)
```

Por tanto

```
sigignore(señal)
```

es equivalente a

```
signal(señal,SIG_IGN)
```

o bien,

```
sigset(señal,SIG_IGN)
```

APÉNDICE A

PROGRAMAS PARA PRÁCTICAS

1.1 Programa sobre sucesiones

El siguiente programa incompleto es la base de las prácticas de los capítulos 6 y 7. Además de inacabado, existen algunos problemas de control del programa que deben resolverse, por ejemplo el programa no termina una vez seleccionada una sucesión.

```
/*****************************************************************/
/*                      Calcular sucesiones                      */
/*****************************************************************/
#include "stdio.h"

extern void fibonacci();
extern void ndnmas1();
extern void armonica();

int main (void)
{
int resp;
int opcion;

      for (resp='n'; resp != 's'; )
      {
            printf ("1-Fibonacci\n");
            printf ("2-n/n+1\n");
            printf ("3-Armonica\n");
            printf ("Elija una sucesion\n");
            scanf ("%d", &opcion);
            switch (opcion)
            {
                  case 1: fibonacci();
                        break;
                  case 2: ndnmas1(1);
                        break;
                  case 3: armonica();
                        break;
                  default:
                        printf ("Opcion incorrecta\n");
                        break;
            }
            printf ("Desea continuar (s/n)? ");
            while ((resp=getchar()) == '\n');
      }
}
```

```
/*****************************************************************/
/*                    Sucesión de Fibonacci                      */
/*****************************************************************/
#define TRUE  1
void fibonacci(void);

void fibonacci(void)
{
int uno, dos, temp;

      uno=0;
      dos=1;
      printf("%d\n",uno);
      printf("%d\n",dos);
      while (TRUE)
      {
            temp=dos;
            dos=uno+dos;
            uno=temp;
            printf ("%d\n",dos);
            sleep(1);                    /* Para visualizar la salida */
      }
}
```

1.2 Programa para cifrar ficheros

```
/*****************************************************************/
/*                  Cifrar la entrada estándar                   */
/*****************************************************************/
#include <stdio.h>
#include <ctype.h>
#include "cifrar.h"
char  fich[NLINEA][NCAR];
extern short menu(int, char *[]);
extern int  leer();
extern void invertircar(int);
extern void invertirlinea(int);
extern void imprimir(int);
main (int argc, char *argv[])
{
int   nlinea;

      switch (menu(argc, argv))
      {
            case 0:
                  nlinea=leer();
                  invertircar(nlinea);
                  invertirlinea(nlinea);
                  break;
            case 1:
                  nlinea=leer();
                  invertirlinea(nlinea);
                  invertircar(nlinea);
                  break;
            default:
                  printf ("Opcion no valida\n");
                  printf("0-Cifrar, 1-Descifrar\n");
                  break;
      }
      imprimir(nlinea);
}
```

```
/*****************************************************************/
/*                    Menú para recoger la opción                */
/*****************************************************************/
short menu(int, char *[]);

short menu(int argc, char *argv[])
{
short opcion;


      if (argc != 2)
      {
            printf ("Utilice: %s valor\n", argv[0]);
            printf("valor=0-Cifrar, 1-Descifrar\n");
            exit(1);
      }
      sscanf(argv[1], "%hd", &opcion);
      return(opcion);
}
```

```
/*****************************************************************/
/*                    Leer la entrada estándar                   */
/*****************************************************************/
#include <stdio.h>
#include "cifrar.h"
extern char fich[NLINEA][NCAR];

int leer();
char *cargar(int);

int leer()
{
int   i;

      for (i=0; cargar(i) != NULL; i++);
      return (i);
}

char *cargar(int linea)
{
      return(fgets(fich[linea], NCAR, stdin));
}
```

```
/*****************************************************************/
/*  Invertir los caracteres de una linea: invertir la línea.  */
/*****************************************************************/
#include "cifrar.h"
extern char     fich[NLINEA][NCAR];

void invertircar(int);
void vueltalin(char []);

void invertircar(int nlinea)
{
int i;

        for (i=0; i<nlinea; i++)
        {
                vueltalin(fich[i]);
        }
}


void vueltalin(char lin[])
{
int i;
int j;
char buftemp[NCAR];

        for (i=strlen(lin)-2, j=0; i>=0; i--, j++)
        {
                if   (islower(lin[i]))
                        buftemp[j]=toupper(lin[i]);
                else if (isupper(lin[i]))
                        buftemp[j]=tolower(lin[i]);
                else
                        buftemp[j]=lin[i];
        }
        buftemp[j]='\0';
        strcpy(lin, buftemp);
}
```

```
/*****************************************************************/
/*           Invertir líneas de la entrada estándar              */
/*****************************************************************/
#include "cifrar.h"

extern char fich[NLINEA][NCAR];
void invertirlinea(int);

void invertirlinea(int nlinea)
{
int   mitad;
int   i,j;
char  buftemp[NCAR];

      mitad=(nlinea-1)/2;
      for (i=0, j=nlinea-1; i <= mitad; i++, j--)
      {
            strcpy (buftemp, fich[i]);
            strcpy (fich[i], fich[j]);
            strcpy (fich[j], buftemp);
      }
}
```

```
/*****************************************************************/
/*          Imprimir el resultado de la transformación           */
/*****************************************************************/
#include "cifrar.h"

extern char fich[NLINEA][NCAR];
void imprimir(int);

void imprimir(int nlinea)
{
int   i;

      for (i=0; i<nlinea; i++)
      {
            printf("%s\n",fich[i]);
      }
}
```

APÉNDICE B
SOLUCIONES

1.1

1.2

1.3 Capítulo 3

1.

```
$ awk '/M*/' notas
```

Mostraría todas las líneas, ya que la cadena nula también cumple M*.

```
$ awk '/^M*/' notas
```

Igual que la anterior.

```
$ awk '/M.*/'notas
```

Sólo se visualizarán las líneas contengan una M.

2.

```
$ awk '!/Marta/' notas
$ awk '$1 !~ /^Marta$/'notas
$ awk '$1 != "Marta" notas
```

3.

```
$ awk 'NR==2,NR==8' notas
$ awk '{if (NR>=2 && NR <=8) print $0}' notas
```

4.

```
$ awk -F# '$2=="Fisica" && max < $3 {
          max=$3
          nombre=$1
          }
          END {print nombre,max}' notas
Pablo 8
```

5.

6.

7.

8.

9.

```
$ cat tabla2
awk -F#   \
        '$2=="Fisica" {
                        alumn["Fisica,1"]+=1
                        alumn["Fisica,2"]+=1
                        alumn["Fisica,3"]+=1
                        if ($3>=5)
                                porce["Fisica,1"]+=1
                        if ($4>=5)
                                porce["Fisica,2"]+=1
                        if ($5>=5)
                                porce["Fisica,3"]+=1
                      }
         $2=="Matematicas" {
                        alumn["Matematicas,1"]+=1
                        alumn["Matematicas,2"]+=1
                        alumn["Matematicas,3"]+=1
                        if ($3>=5)
                                porce["Matematicas,1"]+=1
                        if ($4>=5)
                                porce["Matematicas,2"]+=1
                        if ($5>=5)
                              porce["Matematicas,3"]+=1
                    }
        $2=="Dibujo" {
                      alumn["Dibujo,1"]+=1
                      alumn["Dibujo,2"]+=1
                      alumn["Dibujo,3"]+=1
                      if ($3>=5)
                              porce["Dibujo,1"]+=1
                      if ($4>=5)
                              porce["Dibujo,2"]+=1
                      if ($5>=5)
                              porce["Dibujo,3"]+=1
```

```
                    }
      END           {
                      asig["0"] = "Fisica"
                      asig["1"] = "Matematicas"
                      asig["2"] = "Dibujo"
                      printf "                Exam.1 Exam.2
Exam.3\n"
                      for (a in asig)
                      {
                              printf "%-16s",asig[a]
                              fila=0
                                for (i=1; i<=3; i++)
                                {
                                  porce[asig[a]","i]=
porce[asig[a]","i]/ alumn[asig[a]","i]*100
                                  printf "%6.2f "
,porce[asig[a]","i]
                                  fila+= porce[asig[a]","i]
                                }
                                printf "%6.2f\n", fila/3
                        }
                        printf"-----------------------------------
---------------\n%16s"," "
                      for (i=1; i<=3; i++)
                      {
                              col=0
                              for (a in asig)
                              {
                                      col+= porce[asig[a]","i]
                              }
                              printf "%6.2f ", col/3
                      }
                    }' notas
```

1.4 Capítulo 4

3. Pág. 50

```
man=´who | wc -l´
eval $man
```

4. pág. 50

```
read -r linea
eval "$linea"
```