upna Universidad Pública de Navarra Nafarroako Unibertsitate Publikoa

Área de Lenguajes y Sistemas Informáticos
Departamento de Ingeniería Matemática e Informática

Prácticas de Sistemas operativos

240304 y 250304 – Grado en Ingeniería Informática
Segundo curso, semestre de otoño

CURSO ACADÉMICO 2020-2021

José Javier Astrain Escola
Mikel Aldaz Zaragüeta
Aitor González de Mendívil Grau

La asignatura 240304 - 250304 - *Sistemas Operativos* de tercer semestre del Grado en Ingeniería Informática dispone de tres créditos ECTS asignados a prácticas. Dichas prácticas tendrán lugar en el horario y aula que se detalla a continuación:

Grupo teoría	Grupo prácticas	Profesor	Horario	Aula
G1	G1P1	José Javier Astrain	Miércoles, 19:00 a 21:00	A338
G1	G1P2	Aitor González de Mendívil	Miércoles, 17:00 a 19:00	A338
G2	G2P1	Mikel Aldaz	Jueves, 19:00 a 21:00	A338
G2	G2P2	Aitor González de Mendívil	Miércoles, 19:00 a 21:00	A310

Las prácticas se realizarán sobre PCs virtualizados equipados con sistema operativo Linux. Los equipos virtuales estarán disponibles en horario 24x7 para su uso por los alumnos.

Durante este curso, se realizarán las siguientes prácticas:

Calendario de prácticas (G1P1, G1P2, G2P2 - miércoles)

	Sesión	Contenido
02/09/2020	1	Operaciones de E/S, manejo de ficheros.
09/09/2020	2	Procesos e hilos.
16/09/2020	3	Comunicación entre procesos: pipes.
23/09/2020	4	Comunicación entre procesos: señales.
30/09/2020	5	Comunicación entre procesos: semáforos y memoria compartida.
07/10/2020	6	Comunicación entre procesos: colas de mensajes.
14/10/2020	7	Construcción de una shell.
21/10/2020	8	
28/10/2020	9	
04/11/2020	10	Construcción de un planificador de procesos mediante colas de múltiples niveles.
11/11/2020	11	
18/11/2020	12	
25/11/2020	13	Construcción de un pequeño sistema concurrente.
02/12/2020	14	
09/12/2020	15	

Calendario de prácticas (G2P1 - jueves)

	Sesión	Contenido
03/09/2020	1	Operaciones de E/S, manejo de ficheros.
10/09/2020	2	Procesos e hilos.
17/09/2020	3	Comunicación entre procesos: pipes.
24/09/2020	4	Comunicación entre procesos: señales.
01/10/2020	5	Comunicación entre procesos: semáforos y memoria compartida.
08/10/2020	6	Comunicación entre procesos: colas de mensajes.
15/10/2020	7	Construcción de una shell.
22/10/2020	8	
29/10/2020	9	
05/11/2020	10	Construcción de un planificador de procesos mediante colas de múltiples niveles.
12/11/2020	11	
19/11/2020	12	
26/11/2020	13	Construcción de un pequeño sistema concurrente.
10/12/2020	14	

Para superar el conjunto de la asignatura es imprescindible haber superado la parte práctica de la asignatura. La parte práctica de la asignatura se evaluará mediante la entrega y evaluación de prácticas, que es de carácter no recuperable y tiene un peso del 20% de la calificación, y un examen práctico al final del semestre, que es de carácter recuperable y tienen un peso del 30% de la calificación. Cada alumno entregará para su evaluación tres prácticas (shell, planificador y sistema concurrente) a lo largo del semestre. Las entregas de las prácticas se realizarán mediante la herramienta MiAulario. Las fechas de entrega de las prácticas serán las siguientes:

1) Construcción de una Shell: 30/10/2020.
2) Construcción de un planificador de procesos: 20/11/2020.
3) Construcción de un pequeño sistema concurrente: 11/12/2020.

Aquellos alumnos que no superen el examen práctico de la asignatura mediante la evaluación coninua, podrán realizar el examen de recuperación (50% de la calificación final) en el que ya no se tendrán en cuenta las calificaciones de las prácticas.

Las prácticas que se entreguen para su evaluación deben contener el trabajo original y personal del alumno que las entregue. En el caso de detectarse plagio o incumplirse los requisitos citados previamente, la parte práctica quedará automáticamente suspendida.

Tutorías

Las tutorías tendrán lugar en los lugares y horarios que a continuación se indican.

Profesor	Lugar	Horario
José Javier Astrain	ZOOM	Lunes y miércoles de 9:00 a 11:00h. Miércoles de 17:00 a 19:00h.
Mikel Aldaz	ZOOM	Martes, de 17:00 a 19:00h. Miércoles, de 17:30 a 19:30h. Viernes, de 11:30 a 13:30h.
Aitor González de Mendívil	ZOOM	Miércoles y jueves de 17.00 a 19.00h.

El resto del documento se organiza en cuatro bloques temáticos:

1. Introducción al Sistema Operativo LINUX.
2. Introducción básica al Lenguaje C.
3. Introducción de llamadas al Sistema Operativo LINUX.
4. Guiones de las prácticas de la asignatura.

Introducción al sistema operativo LINUX

Introducción Básica al Sistema Operativo LINUX

ÍNDICE

0.- Consideraciones previas

Este guión tiene como fin guiar al alumno en su familiarización con el sistema LINUX. No se pretende elaborar un manual completo que describa sus características, funciones y utilidades, sino que sea una guía básica de introducción. Esto implica que muchos puntos de los a continuación expuestos pueden resultar incompletos para el lector y para subsanar este problema se le sugiere que emplee el manual *on-line* del propio LINUX y/o consulte las diferentes referencias bibliográficas que se sugieren.

Acceso al manual

El manual de LINUX ofrece una ayuda *on-line* que está dividida en ocho secciones:

Sección 1	Comandos de usuario (*User Commands*).
Sección 2	Llamadas al sistema (*System calls*).
Sección 3	Biblioteca de funciones de C (*C Library Functions*).
Sección 4	Dispositivos e interfaces de red (*Devices and Network Interfaces*).
Sección 5	Formatos de ficheros (*File Formats*).
Sección 6	Juegos y demostraciones (*Games and Demos*).
Sección 7	Varios (*Miscellaneous*).
Sección 8	Referencia del Administrador (*Administrator Reference*).

El manual se encuentra en /usr/man y se accede a él con el formato:

```
man [sección] comando
```

Las páginas solicitadas se formatean con el programa `nroff` y se muestran con la utilidad `more`. Si no se especifica otra opción, se muestra la primera página del manual que se encuentre con ese nombre. Si se especifica la seción del manual en la que se encuentra la información buscada, se mostrará exclusivamente esa referencia.

```
man -k palabra
```

consulta el índice en busca de alguna referencia a la palabra especificada como argumento. El resultado se muestra en un listado de todas las entradas del índice que contienen la palabra buscada.

Partes de una entrada del manual:

Name	Nombre y función. Estas líneas forman el índice que se consulta mediante `man -k`
Synopsis	Diagrama sintáctico: el nombre, seguido de las opciones (entre corchetes) y posibles argumentos. En el caso de llamadas al sistema y rutinas de biblioteca, el formato y tipo de los parámetros.
Description	Breve descripción del comando.
Return Value	En entradas correspondientes a llamadas al sistema y rutinas de biblioteca, lista de los posibles valores de retorno.
Errors	En entradas correspondientes a llamadas al sistema, lista de los posibles errores.
Options	En entradas correspondientes a comandos, lista detallada de las posibles opciones y sus efectos.
Commands	En entradas correspondientes a comandos interactivos, lista detallada de los mandatos propios de la utilidad.
Files	Ficheros relacionados.
See Also	Referencias a otras entradas del manual y otros tipos de documentación.
Diagnostics	En entradas correspondientes a comandos, lista de mensajes de diagnóstico y error que pueden producirse.
Bugs	Problemas conocidos, o cuestiones pendientes de resolver.

1.- Introducción

Historia.

La primera versión de UNIX fue desarrollada por Ken Thompson en los *Laboratorios Bell (AT&T)* en 1969. Se empleó por primera vez sobre una máquina *PDP-7* de *DEC*. Se programó en ensamblador, y cuando Dennis Ritchie desarrolló el lenguaje C, se reescribió en C.

Versiones.

Actualmente se emplean principalmente cuatro versiones diferentes del sistema operativo UNIX, que son:

- UNIX System V distribuido por *AT&T*.
- BSD v. 4.3 distribuido por la Universidad de California en Berkeley.
- SunOS/Solaris distribuido por la empresa *SUN*.
- Linux.

En 1984 comenzó el desarrollo de un sistema operativo similar a UNIX de libre distribución dentro de lo que se dio en llamar el Proyecto GNU. En la actualidad se emplean ampliamente variaciones del sistema de GNU con el kernel de Linux, llamándose sistema Linux/GNU (http://www.gnu.org).

El MIT (Instituto Tecnológico de Massachusetts) distribuye desde 1984, y de forma gratuita, una interfaz gráfica basada en este sistema operativo que se denomina X-Window. Para encontrar más información sobre los recursos disponibles para este sistema operativo o sus condiciones de uso, se recomienda visitar www.linux.org y www.gnu.org .

Por su elevado grado de aceptación y por ser un software de libre distribución en constante evolución con unas prestaciones similares a otros sitemas UNIX, se ha adoptado este sistema operativo como base para la realización de las prácticas correspondientes a la asignatura *Sistemas Operativos*.

2.- Funcionamiento inicial

LINUX es un sistema operativo multitarea, lo que significa que permite la realización de varias tareas concurrentemente. No emplea un único hilo de ejecución con asignación de tiempos de CPU a cada proceso, sino que cada proceso activo tiene su propio hilo de ejecución con una prioridad que puede ser modificada por el usuario, y en función de la cual el sistema realiza la planificación de las tareas.

LINUX es también un sistema operativo multiusuario, lo que significa que permite que más de un usuario utilice simultáneamente los recursos del sistema. Para que esto pueda ser posible, cada usuario debe identificarse antes de comenzar a usar el sistema, es decir, se necesita una *cuenta* en el sistema. La cuenta está formada básicamente por un *nombre de usuario* (*login*)y una *clave de acceso* (*password*). Para poder comenzar a trabajar con el sistema, el usuario deberá introducir su nombre de usuario y su clave. A partir de ahí el sistema lanza un primer programa (que generalmente se conoce como una *Shell* o una consola) que se ejecuta con el identificador y privilegios de ese usuario.

En el laboratorio, cada grupo de prácticas dispondrá de una cuenta con su correspondiente nombre de usuario. Para iniciar la sesión, cada usuario debe introducir por teclado su nombre de usuario ante el mensaje por pantalla: `login:`
Y posteriormente su contraseña tras el mensaje:

`Password:`

Una vez iniciada la sesión, será cada usuario el que elija su propia password, mediante el comando passwd. La *password* debe tener al menos 8 caracteres y al menos un carácter debe ser numérico. Es obvio decir que cada password debe ser celosamente guardada por su dueño para evitar que un intruso entre en el sistema identificándose como él. Si un usuario tuviera acceso a una cuenta de la cual no es titular, podría acceder a la información allí contenida, o incluso borrarla.

Cada usuario tiene asignado un directorio en el sistema de ficheros. Ese directorio consta como propiedad de ese usuario y en él éste tiene permiso para crear/borrar/modificar ficheros o cambiar permisos.

Es conveniente hacer una copia de seguridad de la información más relevante a disco blando al finalizar cada sesión de prácticas.

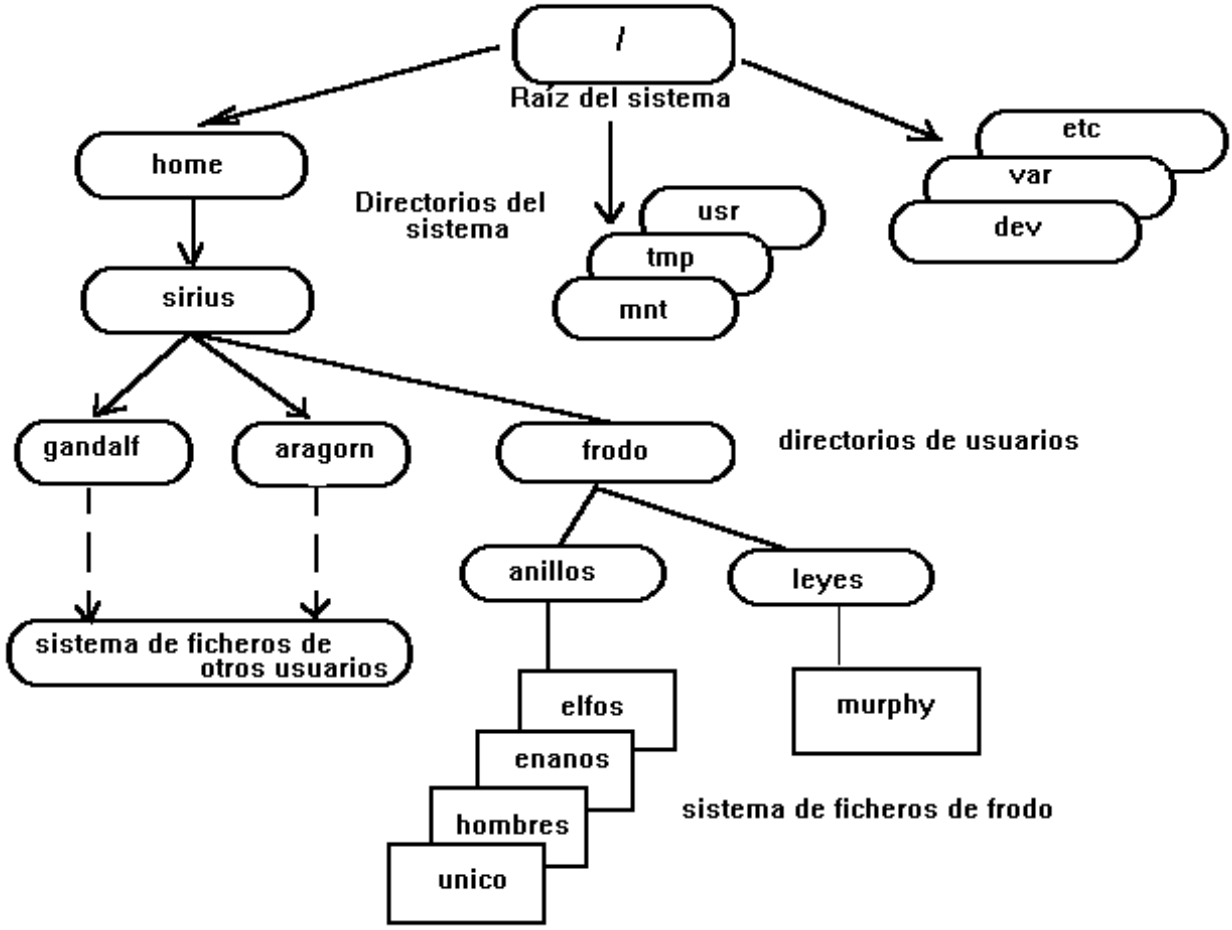
Para concluir la sesión de trabajo hay que cerrar la sesión de trabajo y regresar a la pantalla de identificación inicial del entorno de ventanas.

Nota importante: No apagar NUNCA los ordenadores, aunque SÍ los monitores. LINUX es un sistema multiusuario y multitarea como ya se ha comentado, y por tanto puede haber usuarios utilizando el sistema a través de la red que no desean ver interrumpido su trabajo.

3.- El sistema de ficheros

Un sistema de ficheros proporciona un método adecuado para organizar y almacenar ficheros. Todos los ficheros en LINUX residen en un sistema de ficheros, sin que sea relevante el tipo de los mismos.

El sistema de ficheros en LINUX es una organización de directorios y ficheros estructurada en forma de arbol. La *raíz* de un sistema de ficheros de LINUX se repersenta con el carácter /.



Un fichero puede ser idenificado de forma única especificando la ruta desde la raíz hasta él en el árbol de directorios, en tal caso la ruta empieza en /. Si la ruta no comienza por / se entiende que empieza en el directorio de trabajo del proceso que intenta hacer referencia a ese fichero. Si se desea acceder a un fichero situado en el directorio actual, basta con indicarlo del siguiente modo: *./nombre_del_fichero*. Si el fichero se encuentra en un directorio anterior, se indica: *../nombre_del_fichero* .

Un fichero consiste en una sucesión de bytes terminada por una marca de *fin de fichero*. Físicamente un fichero puede contener una serie de bloques de disco o cinta. Los

bloques de datos que pertenecen a un fichero pueden estar dispuestos de forma aleatoria en el sistema de almacenamiento.

Un directorio no es sino otro tipo más de fichero que puede contener ficheros.

Ficheros ordinarios

Un fichero ordinario contiene datos arbitrarios en cero o más bloques de datos almacenados en un sistema de ficheros. Estos ficheros pueden contener texto ASCII o datos binarios. No existe ninguna estructura impuesta por el sistema operativo sobre cómo se debe organizar un fichero. LINUX no hace ninguna distinción entre ficheros que contienen diferentes tipos de datos.

Directorios

Los directorios son un tipo especial de ficheros que proporcionan la relación entre nombres de ficheros y los ficheros propiamente dichos. Como resultado de esto, la estructura de los directorios define la estructura del sistema de ficheros completo.

Un directorio consiste en una tabla cuyas entradas, una para cada fichero, contienen: un número de *inodo* y un nombre de fichero empleado para hacer referencia, de forma simbólica, a ese *inodo*. Cada entrada en la tabla del directorio se emplea para convertir el nombre de un fichero en su correspondiente *inodo*.

Cada proceso (programa en ejecución) se encuentra siempre en un directorio, es lo que se llama su *directorio de trabajo (working directory),* que el proceso puede cambiar a voluntad con el comando `cd`.

Ficheros especiales

Los ficheros especiales no contienen datos. En vez de eso proporcionan un mecanismo para relacionar dispositivos físicos con nombres de fichero en el sistema de ficheros. Cada dispositivo soportado por el sistema está asociado con al menos un fichero especial. Cuando se realiza una petición de lectura o escritura sobre un fichero especial, resulta en la activación del controlador asociado con ese dispositivo. Este controlador es la parte del código del sistema encargada de controlar las operaciones relacionadas con el dispositivo físico.

Se mencionan a continuación algunas utilidades relacionadas con el manejo de ficheros que pueden resultar de interés:

- *ls*

Lista ficheros. Si no se especifica el directorio se toma el directorio detrabajo del proceso que lo ejecuta (algunas opciones útiles: -alF).

• *cat*

Muestra por pantalla el contenido de un fichero. Todos los ficheros ordinarios en LINUX son similares, es decir, son simplemente un conjunto de bytes. No hay diferencia entre lo que en otros sistemas se conoce como ficheros de texto y ficheros binarios. Sin embargo, ficheros que no contengan texto simple generalmente contendrán bytes con valores que no hagan referencia a caracteres imprimibles. Por lo tanto no es aconsejable utilizar cat (o cualquier otra utilidad para mostrar el contenido de ficheros) sobre ficheros que no son texto simple.

• *cp <origen> <destino>*

Hace una copia de un fichero.

• *mv <origen> <destino>*

Mueve un fichero de un lugar a otro. En realidad elimina del directorio la entrada que hace referencia a ese fichero y añade una nueva en el directorio especificado en el destino. Sirve también para cambiar el nombre que tiene un fichero en un directorio.

• *rm <fichero>*

Elimina la entrada en un directorio referente a un fichero. Con la opcion -r elimina tambíen directorios, aunque estos no estén vacíos.

• *mkdir <nombre>*

Crea un nuevo directorio.

• *rmdir <nombre>*

Elimina un directorio. Es necesario que no contenga ningún fichero.

• *pwd*

Muestra el directorio de trabajo actual.

• *cd <nombre>*

Permite cambiar el directorio de trabajo del Shell.

• *chmod <permisos> <fichero>*

Permite cambiar el conjunto de permisos de un fichero.

Permisos de los ficheros

En LINUX, todos lo ficheros tienen un propietario y un grupo. El propietario es quien lo creó y el grupo generalmente es el grupo al que pertenece el propietario.

Todos los ficheros tienen unos permisos que permiten a unos usuarios u otros realizar ciertas operaciones con ellos. Los permisos están en tres categorías:

- Permisos de propietario: Son los que se aplican al propietario del fichero.
- Permisos de grupo: Se aplican a todos los miembros de ese grupo que no son el propietario.
- Permisos para el resto: Se aplican a todos los que no entran en ninguna de las dos categorías anteriores.

En cada una de estas categorías hay tres permisos:

- Permiso de lectura: Permite leer el fichero.
- Permiso de escritura: Permite modificar el fichero.
- Permiso de ejecución: Permite ejecuarlo.

En el caso en que el fichero sea un directorio, el permiso de lectura permite listar su contenido y el de ejecución permite mover el directorio de trabajo de un proceso a ese directorio. Para eliminar un fichero hace falta permiso de escritura en el directorio que lo contiene (se debe modificar la tabla de ese directorio).

Estos permisos se organizan de la siguiente forma:
Ejemplo:

```
-rwx-w----
```

El primer caracter es el tipo de fichero:
Tipos:

-	ordinario
d	directorio
c	dispositivo
s	fichero para comunicación entre procesos (socket)
l	enlace simbólico

Los siguientes 3 caracteres representan los permisos del propietario, luego vienen los del grupo y finalmente los del resto de usuarios.

Los símbolos son los siguientes:

r	leer
w	escribir
x	ejecutar

Como ya se ha mencionado, la utilidad chmod sirve para cambiar los permisos de un fichero. Tiene fundamentalmente dos modos de empleo:

Ejemplo 1:

```
% chmod 754 fichero
```

Entiende una sintaxis numérica, a cada categoría le asigna un dígito octal, de tal modo que 1 permite el acceso, y 0 lo deniega. El primer dígito representa los permisos del propietario; 7 en binario es 111, lo cual corresponde a los tres permisos activos (lectura, escritura, ejecución). El segundo dígito representa los permisos del grupo; 5 en binario es 101, lo que corresponde a lectura y ejecución activado, escritura desactivado. El tercer dígito son los permisos para el resto del mundo; 4 en binario es 100, sólo permiso de lectura.

Ejemplo 2:

```
% chmod o+r fichero
```

En este caso se especifica mediante una letra qué permisos se desea modificar:

- Clase: u : propietario
  g : grupo
  o : resto
  a : todos

Después se especifica la operación que se desea realizar sobre el permiso:

- Operación: + : añade acceso
  - : elimina acceso
  = : pone permiso

Y a continuación sobre qué permiso se desea actuar:

- Permiso: r : lectura
  w : escritura
  x : ejecución

El ejemplo añadiría permiso de lectura al resto de usuarios.

4.- El Bash-Shell

Cuando nos autentificamos ante el sistema, éste lanza un proceso que ejecuta un primer programa para nosotros. Generalmente este programa es lo que se conoce como una Shell o intérprete de comandos. Es un programa que se dedica a recoger del teclado instrucciones respecto a comandos que deseamos ejecutar y a ejecutarlos. Puede añadir muchas otras facilidades.

El *Shell* que se va a emplear en las prácticas es el *Bash-Shell* (*bash*). Este intérprete de comandos evoluciona de un primer intérprete programado en lenguaje C en la Universidad de California (Berkeley, EEUU) que acompañaba a las diferentes distribuciones de LINUX con el nombre de *C-Shell*. Otro *Shell* que se ha difundido notablemente es el *Bourne Shell* (sh), que tiene su origen en los *Laboratorios Bell* y que distribuye AT&T.

Todo lo que se describe a continuación hace referencia al *Bash-Shell*.

En un comando el orden es el siguiente:

```
% comando opción(es) argumento(s) [redireccionamiento(s)]
```

Cuando el *Shell* ejecuta un comando le asigna una entrada estándar, una salida estándar y una salida de error estándar. Normalmente la entrada es el teclado y las salidas se ofrecen en pantalla. Cuando un programa lee de la entrada estándar y escribe en la salida estándar se dice que dicho programa es un filtro.

El *Bash-Shell* permite redirigir los tres canales estándar (entrada, salida y error) a ficheros. La expresión utilizada para redirigir la salida estándar es la siguiente:

```
comando > fichero
```

Ejemplo:

```
% ls -alF > listado
```

Crea el fichero vaciándolo antes si este ya existía previamente. Si lo que se desea es que la salida del comando se añada al contenido de un fichero, basta con formar el comando de la siguiente manera:

```
comando >> fichero
```

Ejemplo:

```
% echo Fin del listado >> listado
```

Se puede hacer que la entrada del comando sea el contenido de un fichero:

```
comando < fichero
```

Ejemplo:

```
% cat < listado
```

También permite conectar la salida estándar de un comando con la entrada estándar de otro; esto es lo que se denomina una *pipe*. Se pueden especificar varias *pipelines* en una sola línea. Para indicar al *Shell* que se desea hacer esta conexión se emplea el carácter `|`.

Ejemplo:

```
ls | wc -l
```

`ls` lista los ficheros de un directorio y envía mediante la pipe esa salida a la utilidad `wc` que con la opción `-l` cuenta el número de líneas.

Comandos incluidos en el Shell

- `cd` Permite cambiar el directorio de trabajo del Shell
- `echo args` Muestra sus argumentos por la salida estándar.

Variables

Los *Shells* soportan dos tipos de variables: variables locales y variables de entorno. Ambos tipos de variables almacenan datos en forma de una cadena. La diferencia principal entre ambos es que cuando el *Shell* crea otro *Shell* (ejecutando /bin/csh), el hijo tiene una copia de las variables de entorno del padre, pero no de las locales.

Cada Shell tiene un conjunto de variables de entorno predefinidas, generalmente en ficheros de inicialización, así como variables locales.

Para acceder al contenido de las variables del Shell hay que colocar $ delante del nombre de las mismas.

Comandos del C-Shell relacionados con las variables:

- `set` Lista las variables definidas.
- `set var = cont` Permite crear la variable `var` y darle por valor `cont`.
- `unset var` Destruye la variable `var`.
- `echo $var` Muestra el contenido de la variable `var`.
- `printenv` Lista las variables de entorno definidas
- `setenv var cont` Permite crear la variable `var` y darle por valor `cont`.
- `unsetenv var` Destruye la variable `var`.

Variables de entorno comunes:

Nombre	Significado
HOME	El directorio del usuario, el camino completo.
PATH	Lista de directorios donde el Shell buscará los comandos que se le pida ejecutar. Si no los encuentra ahí, devuelve error.
USER	Identificador del usuario.
SHELL	Shell en uso, el camino completo.
TERM	Tipo de terminal en uso.

Se puede emplear símbolos para expandir nombres de ficheros, de modo que se simplifique el manejo de éstos. Algunos son:

`*`	Cualquier carácter
`?`	Un único carácter
`[car1...carn]`	Cualquier carácter de la lista o rangos incluidos entre los corchetes.
`{cadena,...}`	Cada una de las cadenas de la lista.
`~usuario`	Directorio inicial del usuario especificado. Si no se especifica el usuario, sino solo el ~ (Altgr+ñ), se entiende que hace referencia al directorio del usuario propietario del proceso Shell.

Historia de comandos

Podemos mantener una historia de eventos que nos permita emplearlos sin tener que teclearlos de nuevo. Para ello hay que crear una variable de entorno del Shell y darle como valor el número de eventos (comandos) que queremos que recuerde (que serán los últimos). Esta variable se llama history. Existe un comando de igual nombre (history) que lista todos los eventos almacenados. Si deseamos repetir uno de esos eventos tenemos varias posibilidades; algunas de las más típicas se listan a continuación:

• `history`	Presenta el listado de eventos.
• `set history` = nº_de_eventos	Establece el número máximo de eventos que serán conservados.
• `!!`	Evento previo.
• `!n`	Evento n-ésimo.
• `!cad`	Evento más reciente que comience con la cadena `cad`.

Ficheros del Shell

El Shell puede ser personalizado por el usuario. Se configura por medio de tres ficheros:

- `.bashrc`
- `.bash_profile`
- `.login`
- `.logout`

Como se puede observar los tres ficheros comienzan con un punto, por lo tanto si queremos listarlos deberemos emplear la opción -a en el comando ls.

El Shell lee al comienzo el fichero .bashrc y lo ejecuta, luego hace lo mismo con el fichero .login. Al finalizar la sesión lee y ejecuta .logout. Tanto .login como .logout sólo se ejecutan una vez, aunque después se activen otros shells estos ficheros no se vuelven a ejecutar.

5.- Caracteres especiales

Algunos caracteres son interpretados de forma especial al ser tecleados en un terminal. Suelen llamarse metacaracteres y se pueden listar con la utilidad stty (`stty -a`).

Ejemplo de parte del resultado:

```
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>; eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
```

El carácter ^ ante una letra significa que se ha de pulsar la tecla *Ctrl* al mismo tiempo que esa letra. El significado de algunos de estos caracteres es:

- intr Termina la ejecución de un proceso.
- eof Su significado es *Fin de Fichero* , en los casos en que se emplea la entrada estándar para dar información a un comando sirve para enviar este carácter que da por finalizada la información.
- erase Corresponde al carácter que provoca el borrado de la letra enterior a la posición del punto de inserción.

6.- Utilidades generales de LINUX

El número de utilidades existentes para LINUX nos obliga a hacer un resumen muy básico.

`grep`	Busca una cadena en uno o más ficheros (o en la entrada estándar) Ej. `grep <cadena> <fichero>`
`head`	Permite ver las primeras líneas de un fichero (o de la entrada estándar). Sin opciones muestra las 10 primeras.
`tail`	Permite ver las últimas líneas de un fichero (o de la entrada estándar). Sin opciones muestra las 10 últimas.
`more`	Permite la observación pausada de un fichero (o de la entrada estándar). Pulsando la barra espaciadora avanza una página, con `b` o `^B` retrocede una página, con `q` termina.
`wc`	Cuenta el número de bytes, palabras o líneas en un fichero (o en la entrada estándar).
`who`	Muestra los usuarios que en este momento están trabajando con el sistema.
`df`	Permite saber cuánto espacio libre hay en cada sistema de ficheros.
`du`	Dice cuántos bloques ocupa un directorio con todos sus ficheros y subdirectorios.
`find`	Realiza una búsqueda recursiva, comenzando por el directorio especificado y descendiendo por los subdirectorios. Ej.: `find / -name mifichero -print` Buscaría desde el directorio raíz ficheros con el nombre `mifichero` y sacaría los resultados por pantalla.

Editor `vi`

Es un programa editor de ficheros de pantalla que no necesita el entorno de ventanas X-Window. Tiene varios modos de funcionamiento, entre los cuales cabe destacar el modo de inserción de texto y el modo de comandos. Para pasar del modo de inserción al modo de comandos se presiona la tecla *Esc.*

Destacaremos las funciones más básicas del modo de comandos:

a	Añadir texto (pasa al modo de inserción y todo lo que tecleemos se añadirá al fichero) tras el carácter sobre el que se halla el cursor.
i	Pasa al modo de inserción e inserta texto delante del carácter sobre el que se encuentra el cursor.
ZZ	Graba el fichero y sale del editor.
:x	Graba el fichero y sale del editor.
:w	Graba el fichero sin salir del editor.
:w fichero	Graba en el fichero con el nombre indicado.
:q	abandona el editor.
:q!	Abandona el editor sin grabar.
x	Elimina el carácter sobre el que está el cursor.
dd	Elimina la línea sobre la que está el cursor.
numdd	Elimina las num líneas situadas tras el cursor.
dw	elimina la palabra sobre la que está el cursor.
u	Deshace la última operación.

Utilidades de red

Estas utilidades nos permiten compartir los recursos de la red. Nuestra máquina puede ver incrementada su potencia y versatilidad gracias a los recursos de otra máquina que pertenezca a nuestra misma red.

`telnet` Permite acceder a otros sistemas. El formato es:

```
telnet hostname
```

Seguramente la máquina nos responderá con su petición de `login:`

`ftp` Permite la transferencia de ficheros entre sistemas en red aunque tengan distintos sistemas operativos. El formato de inicio es: ftp hostname. Con get nos traemos un fichero de una máquina remota y con put llevamos un fichero a la máquina remota. Con help obtenemos una lista de los comandos disponibles en ese servidor.

7.- Conclusiones

Se recomienda explorar el sistema, en especial la información disponible mediante la utilidad man. Existen numerosos libros sobre LINUX que pueden resultar de gran utilidad a la hora de familiarizarse con el sistema, se recomienda acudir a ellos.

8.- Bibliografía

UNIX For Programmers And Users A Complete Guide, G. Glass, Ed. Prentice Hall, ISBN 0-13-061771-7

Advanced Programming In The UNIX Environment, W. Richard Stevens, Ed. Addison-Wesley, ISBN 0-201-56617-7

Beggining Linux Programming, N.Matthew & R.Stones, Ed.Wrox, ISBN 1-874416-68-0

Introducción básica al Lenguaje

C

INDICE

1. INTRODUCCIÓN

C es un lenguaje de programación desarrollado por Dennis Ritchie para codificar el sistema operativo UNIX. Las primeras versiones de UNIX se escribieron en ensamblador, pero a partir de 1973 pasaron a escribirse en C. Sólo un pequeño porcentaje del núcleo de UNIX se sigue codificando en ensamblador; en concreto, aquellas partes íntimamente relacionadas con el hardware. Todas las órdenes y aplicaciones estándar que acompañan al sistema UNIX están escritas también en C. Esta es la razón que hace de este lenguaje la forma natural de comunicarse con el Sistema Operativo UNIX.

En UNIX también se puede programar con otros lenguajes, pero a la hora de desarrollar aplicaciones enfocadas a aprovechar al máximo los recursos del sistema es conveniente tomar la decisión de escribirlas en C.

El lenguaje se puede clasificar dentro del grupo de los lenguajes de alto nivel, aun a pesar de no estar fuertemente tipado (la comprobación que hace de los tipos por ejemplo en una asignación no es muy estricta); sin embargo, también le ofrece al programador posibilidades que sólo están presentes en los lenguajes de bajo nivel. Así, por ejemplo, en C vamos a poder manipular bits y aritmética de direcciones. C también permite el desarrollo de la programación estructurada y modular.

2. CICLO DE CREACIÓN DE UN PROGRAMA

2.1. Introducción

A la hora de crear un programa, hemos de empezar por la edición de un fichero que va a contener el código fuente. Este fichero se nombra, por convenio, añadiéndole la extensión ***".c"*** y se puede editar con cualquier editor de textos que no inserte extraños caracteres de control.

El compilador (*cc o gcc*) es el encargado de generar el fichero ejecutable a partir del fichero fuente. Para invocarlo debemos escribir:

% gcc programa.c

Esta línea de órdenes va a provocar que se genere el fichero *a.out*, que ya es ejecutable. Si queremos que nuestro programa ejecutable tenga algún nombre en concreto, emplearemos la línea:

% gcc -o programa programa.c

lo que va a provocar que el fichero ejecutable se llame *programa*, en lugar de *a.out*.

Gcc no es realmente el compilador, sino un programa estándar que se encarga de invocar al compilador. Los pasos que se ven involucrados en la compilación del *programa.c* quedan reflejados en la figura 1, y son los siguientes:

- Preprocesamiento (cpp). Esto genera el fichero *programa.i*. Acepta como entrada el código fuente, elimina los comentarios e interpreta las directivas especiales del preprocesador que empiezan por #.

- Compilación y optimización (comp). Esto genera el fichero *programa.s* que contiene el código ensamblador.
- Generación del código (as). Esto va a generar el fichero objeto *program.o*.
- Enlace del código objeto con otros módulos objeto y librerías (*ld*). Es en esta fase donde se va a generar el código ejecutable. Si el código fuente hacía referencia a funciones de librerías o funciones definidas en otros códigos fuente, o a variables externas se resuelve aquí.

Como los ficheros intermedios generados por el procesador y el compilador no interesan, se borran al terminar el proceso global.

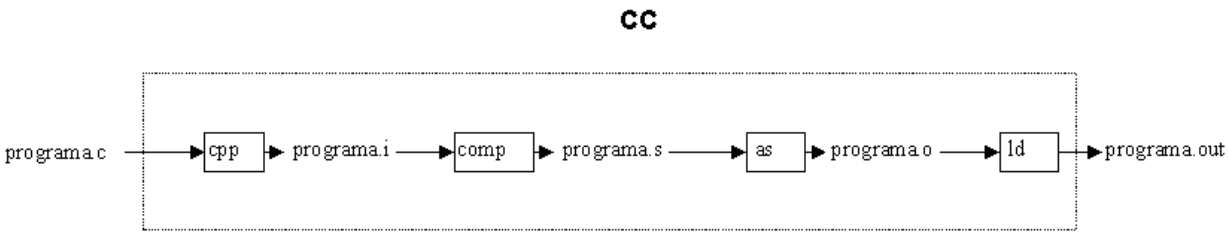


Figura 1: Proceso completo de compilación.

2.2. Módulos

Para la creación de un programa de gran tamaño es recomendable separar el código en módulos que tengan cierta entidad. De esta forma un módulo puede compilarse como librería (extensión *.o*) y poder ser así utilizado por otros programas. Toda librería que se quiera incluir en otro programa ha de acompañarse de un archivo de cabecera (extensión *.h*) que como veremos posteriormente es lo que se incluye mediante la sentencia *#include* en el programa que la utiliza. El archivo de cabecera ha de contener la declaración de las funciones que se encuentran en la librería. También se suele incluir en los archivos de cabecera la declaración de nuevos tipos y constantes.

2.3. Compilación

Para los sistemas UNIX, los compiladores más habituales de C son ***cc*** y ***gcc***. El compilador *gcc* es de libre distribución (bajo licencia GNU, http://www.gnu.org) y se encuentra disponible en *LINUX.*

Las opciones típicas para crear un archivo ejecutable son:

gcc -o <fichero_ejecutable> <fichero_codigofuente> <libreria1> <libreria2>...

Si las librerias son las genéricas del sistema, no hace falta ponerlas y el compilador se encarga de buscarlas, excepto para algunas como la librería matemática <math.h> para la que hace falta añadir -lm en la linea de compilación..

Para crear una librería vale con:

gcc -c <fichero_codigofuente>

Esto crea un fichero objeto acabado en .o que podrá ser utilizado como librería. Para ello en el código fuente principal será necesario hacer un #include de su archivo de cabecera y colocar el fichero *.o* como librería a la hora de compilar el programa.

2.4. Utilidad Make

La utilidad *make* también nos puede servir para generar programas ejecutables a partir de las reglas definidas en un fichero denominado *Makefile*. Por ejemplo, si queremos crear el programa *ejemplo* a partir del fichero fuente *ejemplo.c*, bastará con crear un fichero denominado Makefile que contenga:

```
ejemplo: ejemplo.c
        gcc -o ejemplo ejemplo.c
```

Nota: delante del gcc debe haber exactamente un tabulador, nada más, tampoco valen espacios.

Y la compilación se producirá escribiendo:

% make

El *make* antes de hacer la compilación verifica si se han cambiado alguno de los ficheros y compila sólo aquellos que hayan cambiado.

La estructura general de un Makefile es la siguiente:

```
objetivo: dependencias
        <linea del comando para conseguir ese objetivo a partir de las dependencias>
```

Delante de la linea del comando debe haber un tabulador necesariamente. Por ejemplo, veamos cómo compilar programas que se componen de varios módulos fuente (librerias). Si por ejemplo se tiene un programa compuesto por los siguientes ficheros:

- Dos ficheros fuentes de lenguaje C: *main.c, ejem.c*
- Una librería *lib.o*, con su fichero de cabecera *lib.h* incluído en ambos ficheros de lenguaje C.

Se debe construir un fichero denominado *Makefile* o *makefile* que contenga las sentencias:

```
ejemplo:main.o ejem.o lib.o
                gcc -o ejemplo main.o ejem.o lib.o
main.o:  main.c lib.h
                gcc -c main.c
ejem.o:  ejem.c lib.h
                gcc -c ejem.c
```

Para crear el programa ejecutable únicamente sería necesario hacer como antes:

% make ejemplo o % make

Sin embargo, si el fichero anterior no se llama *Makefile o makefile* sino que se llama de otra forma distinta, *prueba* por poner algún nombre, para crear el programa ejecutable será necesario el siguiente paso:

```
% make -f prueba
```

Si se quiere usar el comando *make* para compilar varios programas que son independientes hace falta decírselo insertando en la primera línea una como la siguiente:

```
all: <objetivo1> <objetivo2> <...>
```

3. COMPONENTES LÉXICOS DEL LENGUAJE C

Existen seis clases de componentes léxicos: identificadores, palabras reservadas, constantes, cadenas de caracteres, operadores y otros separadores.

- Identificador. Es una secuencia de letras y dígitos donde el primer elemento debe ser una letra, o los caracteres '_' y '$'. Las letras mayúsculas y minúsculas se consideran distintas. En toda implementación deben ser significativos al menos los 32 primeros caracteres de un identificador.
- Palabras reservadas. Las siguientes palabras no se pueden usar como identificadores:

auto	extern	sizeof
break	float	static
case	for	struct
char	goto	switch
const	if	typedef
continue	int	union
default	long	unsigned
do	register	void
double	return	volatile
else	short	while
enum	signed	

- Constantes. Consideramos 4 tipos: enteras, de carácter, flotantes y de enumeración.
  Ejemplos:
  - Enteras: 120, -10(número entero)
  - De carácter: 'a', '\n', '\t', '\xfa', '\0', '\033'
  - Flotantes: 123.42, 1.7E-6, 3.45e10
  - Enumeración: enum boolean {NO, SI};
- Cadena de caracteres. Secuencia de caracteres alfanuméricos entre comillas dobles.
  Ejemplo: "Esto es una cadena de caracteres".
- Operadores. Caracteres para identificar los operadores unarios, binarios y ternarios.
  Ejemplo: +, *, &&
- Otros separadores. {, }, [,], (,), ;, ->, .
- Comentarios. Secuencia de caracteres que se inicia con /* y termina con */. Los comentarios no se pueden anidar. Son ignorados por el compilador.

4. ESTRUCTURA DE UN PROGRAMA C

Aunque las estructuras de un programa C pueden ser muy variadas y no hay normas fijas, suele ser común organizar un programa como sigue:

```
# directrices para el preprocesador
declaración de variables y funciones externas
declaración de variables globales y funciones prototipo
funciones (la función main debe aparecer en un módulo, y
sólo en uno)
```

Todo programa C, desde el más pequeño hasta el más complejo, tiene un *programa principal* que es con el que se comienza la ejecución del programa. Este programa principal es también una función, pero una función que está por encima de todas las demás. Esta función se llama **main()** y tiene la forma siguiente:

```
void main(void)
{
sentencia_1
sentencia_2
...
}
```

Las *llaves* {...} constituyen el modo utilizado por el lenguaje C para agrupar varias sentencias de modo que se comporten como una sentencia única (*sentencia compuesta* o *bloque*). Todo el cuerpo de la función debe ir comprendido entre las llaves de apertura y cierre.

Las directrices del preprocesador son órdenes que ejecuta el preprocesador (cpp) para generar el *fichero.i* con el que va a trabajar el compilador. Hay dos directrices que se emplean masivamente en los programas: *#include* y *#define*.

- *#include* se emplea para indicar un *fichero de cabecera* donde están definidos tipos derivados y funciones prototipo. Si es una librería genérica se usa como *#include <nombre_libreria>* y si no lo es como *#include "ruta _libreria"*.
- *#define* se emplea para declarar identificadores que van a ser sinónimos de otros identificadores o constantes. También se emplea para declarar macros.

El siguiente ejemplo nos ayudará a comprender cómo es el aspecto que tiene un fichero fuente C.

Programa 1 Primer programa de ejemplo (ejemplo.c)

```
/***
        PROGRAMA; ejemplo.c
        DESCRIPCIÓN; Primer ejemplo de programa C.
***/
#include <stdio.h> /* Fichero de cabecera para la librería estándar de E/S */
#define VALOR_INICIAL 0
#define VALOR_FINAL 10
```

```
#define INCREMENTO 1

/* Función principal. */
main ()
{
        int i;

        i = VALOR _INICIAL;
        while (i < VALOR_FINAL)
        {
                printf ("i = %d\n",i);
                i = i+1;
        }
}
```

5. TIPOS DE DATOS

En C se consideran dos grandes bloques de tipos de datos: los que suministra el lenguaje (tipos fundamentales) y los que define el programador (tipos derivados).

5.1. Tipos fundamentales

Los tipos fundamentales se clasifican en enteros y reales. Los primeros se utilizan para representar subconjuntos de los número naturales (N) y enteros (Z). Los segundos se emplean para representar un subconjunto de los números racionales (Q).

5.1.1. Tipos enteros

Para declarar variables de alguno de los tipos enteros emplearemos las palabras reservadas *char, int, long, short* y *enum*.

- *char*: define un número entero de 8 bits. Su rango es [-128, 127]. También se emplean para representar el conjunto de caracteres ASCII.
- *int*: define un número entero de 16 ó 32 bits (dependiendo del procesador). Su tamaño suele coincidir con el del bus de datos del procesador.
- *long*: define un número entero de 32 bits ó 64 bits.
- *short*: define un número entero de tamaño menor o igual que int.

Se ha de cumplir que tamaño(short) ≤ tamaño(int) ≤ tamaño(long)

Estos cuatro tipos pueden ir precedidos del modificador unsigned para indicar que el tipo sólo representa números positivos o el cero. Ejemplos:

```
int hora;
char carácter;
unsigned short mes;
```

- *enum*: se utiliza para definir un subconjunto dentro del conjunto de los números enteros. A este subconjunto se le conoce como enumeración y a cada uno de sus elementos se le asocia un identificador. La definición de un tipo enumerado es:
  enum tipo_nonumerado {identificador$_1$, identificador$_2$, ..., identificador$_n$};

Por ejemplo:

```
enum meses {enero=1, febrero, marzo, abril, mayo, junio, julio, agosto, septiembre,
octubre,        noviembre, diciembre};
enum meses mes;          /* mes es una variable declarada del tipo enum meses. */
```

5.1.2. Tipos reales

Para declarar variables de alguno de los tipos reales emplearemos las palabras reservadas *float* y *double*.

- *float*. define un número en coma flotante de precisión simple. El tamaño de este tipo suele ser de 4 bytes (32 bits).
- *double*: define un número en coma flotante de precisión doble. El tamaño de este tipo suele ser de 8 bytes (64 bits). El tipo *double* puede ir precedido del modificador *long*, lo que indica que su tamaño pasa a ser de 10 bytes.

Ejemplos:

```
float temperatura =36.5;
double distancia = 128e64;
```

5.1.3. Tipo void

La expresión *void* denota un valor inexistente y se utiliza, por ejemplo, para ignorar el valor devuelto por una función. El valor de un objeto *void* no se puede utilizar en ninguna forma, ni se puede aplicar la conversión de un *void* a otro tipo *no void*.

Por otro lado, cualquier apuntador se puede convertir a tipo *void* * sin pérdida de información. Por tanto, se puede utilizar *void* * como puntero genérico en asignaciones y relaciones.

5.2. Operador sizeof

Para determinar el tamaño, en bytes, tanto de un tipo fundamental como de uno derivado, podemos usar el operador sizeof. Esto ayudará a que nuestros programas sean más transportables entre máquinas donde el tamaño de los tipos no coincida.

El siguiente ejemplo es un programa que muestra el tamaño de cada uno de los tipos fundamentales.

Programa 2 Impresión del tamaño de los tipos fundamentales (tipos.c)

```
/***
```

```
        PROGRAMA; tipos.c
        DESCRIPCIÓN: Este programa presenta por pantalla el tamaño de los tipos
        fundamentales de C.
***/
#include <stdio.h>
main ()
{
                printf  ("TIPOS BASE DE C.\n");
                printf  ("char\t\t==>\t%d bytes\n", sizeof (char));
                printf  ("unsigned char\t==>\t%d bytes\n", sizeof (unsigned char));
                printf  ("short\t\t==>\t%d bytes\n", sizeof (short));
                printf  ("unsigned short\t==>\t%d bytes\n", sizeof (unsigned short));
                printf  ("int\t\t==>\t%d bytes\n", sizeof (int));
                printf  ("unsigned int\t==>\t%d bytes\n", sizeof (unsigned int));
                printf  ("long\t\t==>\t%d bytes\n", sizeof (long));
                printf  ("unsigned long\t==>\t%d bytes\n", sizeof (unsigned long));
                printf  ("float\t\t==>\t%d bytes\n", sizeof (float));
                printf  ("double\t\t==>\t%d bytes\n", sizeof (double));
                printf  ("long double\t\t==>\t%d bytes\n", sizeof (long double));
}
```

5.3. Tipos derivados

Los tipos derivados se construyen a partir de los tipos fundamentales o de otros tipos derivados. Vamos a estudiar los siguientes: arrays, punteros, estructuras, uniones y campos de bits.

5.3.1. Arrays

Los arrays son bloques de elementos del mismo tipo. El tipo base de un array puede ser un tipo fundamental o un tipo derivado. Los elementos individuales del array van a ser accesibles mediante una secuencia de índices. Los índices, para acceder al array, deben ser variables o constantes del tipo entero. Se define la dimensión de un array como el total de índices que necesitamos para acceder a un elemento particular del array.

La definición formal de un array N-dimensional es la siguiente:

tipo_array nombre_array [rango$_1$][rango$_2$]...[rango$_N$] ;

Ejemplos:

```
  int matriz_entera [10][10];   /* Matriz de números enteros de 10 filas por 10 columnas
*/
  int vector_real [5];          /* Array unidimensional de 5 elementos. */
```

A los array unidimensionales se les llama *vectores*, y a los bidimensionales, *matrices*.

Para indexar los elemento de un array, tendremos en cuenta que los índices deben variar entre 0 y M-1, donde M es el tamaño de la dimensión a la que se refiere el índice.

Para el *array vector_real* declarado antes, el índice variará entre 0 y 4: *vector_real[0], vector_real[1],....., vector_real[4].*

Los vectores de tipo char se conocen como *cadenas de caracteres (strings)*, y los arrays de cadenas de caracteres (matrices de caracteres) se conocen como *tablas*.

Ejemplo:

```
char texto[50];          /* Cadena de 50 caracteres */
char texto[40]="Texto contenido";  /* Cadena de caracteres que se inicializa en la declaración */
        char texto[]="Texto contenido";        /* Si no le damos el tamaño lo saca de la inicialización */
char tabla[50][20];              /* Tabla de 50*20 caracteres */
```

Internamente la cadena de caracteres termina en un carácter nulo '\0', de forma que cuando se procesa se puede saber cual es el final. Por tanto, si queremos almacenar 10 caracteres será necesario reservar un tamaño de string de 11 caracteres, para que el último esté ocupado por '\0'. Para el manejo de cadenas de caracteres existen funciones como *strcpy* y *strcmp*.

5.3.2. Punteros

Los punteros son variables que pueden almacenar direcciones de memoria. Se definen también en base a un tipo fundamental o a un tipo derivado. La declaración de un puntero es como sigue:

*tipo_base *puntero*;

Ejemplo:

```
float *x;
```

X es una variable que contiene la dirección de memoria donde se encuentra un número del tipo *float*. Hay que poner especial atención para no confundir la dirección a la que apunta el puntero con lo que hay en esa dirección.

Para trabajar con punteros hay definidos dos operadores unarios: & y *. El operador & da la dirección de memoria asociada a una variable y se utiliza para inicializar un puntero. El operador * se utiliza para referirse al contenido de una dirección de memoria.

Ejemplo:

```
float x, *px;
px = &x;
*px = 10.05; /* Esta línea de código es equivalente a x = 10.05. A través de px se
        puede acceder de una forma indirecta a la variable x. */
```

El acceso a una variable a través de un puntero se conoce también como indirección. Hay que tener presente que antes de manipular el contenido de un puntero hay que inicializar el puntero para que apunte a una zona de memoria correcta.

Los punteros admiten las operaciones de incremento, decremento, suma de una constante entera y diferencia de punteros. Al realizar estas operaciones, realmente estamos modificando la dirección a las que apunta el puntero. No se debe confundir la aritmética de las direcciones utilizando punteros con la aritmética usando variables de tipo entero. Una dirección no es un tipo entero.

Ejemplo:

```
int  array [10];
int *p = array;
...
*p = 5; /* Equivale a array [0] = 5. */
p = p+2;
*p = 3; /* Equivale a array [2] = 3. */
```

El ejemplo anterior muestra cómo podemos acceder a los elementos de un array a través de un puntero. La equivalencia entre arrays y punteros es tan grande que a la hora de indexar los elementos de un array, también podemos hacerlo con punteros.

Ejemplo:

```
int array [10], *p = array, i = 4;
...
*(p+i) = 7; /* Esto equivale a array [i] = 7. */
p[i] = 9; /* Esto equivale a array [i] = 9. */
```

Sin embargo, un array no es lo mismo que un puntero. Mientras que el nombre de un array es una constante (la dirección de inicio del array), un puntero es una variable que puede tomar como valor cualquier dirección.

Uno de los inconvenientes que plantea el uso de arrays es que sus dimensiones deben ser conocidas para el compilador. Los punteros nos ayudan a solucionar este problema, ya que posibilitan el uso de arrays dinámicos (arrays que se dimensionan en tiempo de ejecución). Para hacer esto posible, necesitamos valernos de funciones como *malloc*, que reservan memoria según las necesidades del programa.

Ejemplo:

```
int *p;
...
p = (int *) malloc (20*sizeof(int));/* Esta llamada reserva memoria para 20 números
        enteros a los que vamos a poder acceder a través del puntero p. */
if (p== NULL)
{
        /* Error en la reserva de memoria. Tratamiento del error. */
}
free(p);                /* Libera la zona de memoria reservada en el malloc anterior */
```

La función *malloc* reserva una zona de memoria devolviendo un puntero a *void*, por eso hay que hacer un ***cast*** al tipo que necesitemos, *int* en el ejemplo anterior. Otra

función de reserva de memoria es *calloc,* que además inicializa la memoria a 0. La función *free* libera el espacio de memoria asignado por un *malloc* o *calloc*.

5.3.3. Estructuras

Una estructura es un agregado de tipos fundamentales o derivados que se compone de varios campos. A diferencia de los arrays, cada elemento de la estructura puede ser de un tipo diferente. La forma de definir una estructura es:

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

Ejemplo:

```
struct fecha
{
        unsigned short día;
        unsigned short mes;
        unsigned int año;
};
struct fecha hoy;
```

La variable *hoy* es una variable declarada del tipo *struct fecha*.

Para acceder a los campos de una estructura utilizaremos el operador '.'.Así, para acceder al campo mes de la variable anterior, escribiremos:

hoy.mes = 12;

Se pueden declarar arrays de estructuras:

Ejemplo

```
struct
{
        float real, imaginaria;
}vector[10];

struct Tvector
{
        float real, imaginaria;
};
struct Tvector vector[10];      /* Equivale a la declaración superior */

typedef struct Tvector
{
        float real, imaginaria;
} tipo_vector;
```

```
tipo_vector vector[10];        /* Equivale a la declaración superior */
```

La variable vector es un array unidimensional de número complejos. Para acceder a la parte real del elemento 5, escribiremos:

vector[4].real = 10;

También se pueden declarar punteros a estructuras. En estos casos, el acceso a los campos de la variable se hace por medio del operador '->'. Ejemplo:

```
struct Talumno
{
        char nombre [61];
        float nota;
};
struct Talumno alumno, *pa;
...
pa = &alumno;
pa-> nota = 10.0;                    /* Equivale a alumno.nota=10.0 */
strcpy(pa->nombre, "Gepetto");       /* Equivale a strcpy(alumno.nombre,"Gepetto"); */
```

La indirección *pa->nota* equivale a *(*pa).nota*.

Por último, diremos que puesto que el tipo de cada campo de una estructura pueden ser un tipo fundamental o derivado, también puede ser otra estructura. Tendremos así declaradas estructuras dentro de estructuras.

Ejemplo:

```
struct Tfecha
{
        int día, mes, año;
};
struct Talumno
{
        char nombre [61];
        struct Tfecha fecha_nacimiento;
        float nota;
};
struct Talumno alumno;
```

Con estas declaraciones podremos hacer asignaciones como:

alumno.fecha_nacimiento.mes = 12;

Ejemplo: para realizar una lista de nodos encadenada podemos preparar la siguiente estructura, en la que además de un dato se almacena un puntero que se ha de inicializar apuntando al siguiente nodo de la lista.

```
struct Tnodo
{
        int dato;
        struct Tnodo *siguiente;
}
```

5.3.4 Uniones

Las uniones se definen de forma parecida a las estructuras y se emplean para almacenar en un mismo espacio de memoria variables de distintos tipos. La declaración de una unión es la siguiente:

```
union nombre_union
{
        tipo1 campo1 ;
        tipo2 campo2 ;
        ...
        tipon campon ;
};
```

El tamaño de la unión no va a ser igual a la suma de cada uno de sus campos, como ocurre con las estructuras, sino que es igual al tamaño del mayor de sus campos.
Ejemplo:

```
union número_mixto
{
        float real;
        int entero;
};
union numero_mixto número;
```

El número de esta unión es de 4 bytes (tamaño de campo *real*). Con esta declaración vamos a poder hacer asignaciones como;

```
numero.entero = 10;
numero.real = 125e2;
```

Hay que insistir que la unión no ocupa el espacio de sus dos campos, sino el del mayor de ellos. Si después de las asignaciones anteriores imprimimos el valor de numero.entero, veremos que no es 10.

5.3.5. Campos de bits

El lenguaje C brinda la posibilidad de definir variables cuyo tamaño en bits puede no coincidir con un múltiplo de 8. La forma de declararla es:

```
struct nombre_campo {
        unsigned campo1:tamaño1;   /*tamaño1 indica el número de bits del campo*/
        unsigned campo2:tamaño2;
        ...
        unsigned campon:tamañon;
};
```

Ejemplo:

```
struct byte
{
        unsigned char b0:1;
        unsigned char b1:1;
        unsigned char b2:1;
```

```
        unsigned char b3:1;
        unsigned char b4:1;
        unsigned char b5:1;
        unsigned char b6:1;
        unsigned char b7:1;
};
union BYTE
{
        unsigned char byte;
        struct byte campo;
};
union BYTE byte_1, byte_2;
...
byte_1.byte = 0x76;          /*124 en decimal. Una constante hexadecimal consiste
        en 0x seguido de la constante en forma hexadecimal.*/
byte_2.campo.b0 = byte_1.campo.b7;
byte_2.campo.b1 = byte_1.campo.b6;
byte_2.campo.b2 = byte_1.campo.b5;
byte_2.campo.b3 = byte_1.campo.b4;
byte_2.campo.b4 = byte_1.campo.b3;
byte_2.campo.b5 = byte_1.campo.b2;
byte_2.campo.b6 = byte_1.campo.b1;
byte_2.campo.b7 = byte_1.campo.b0;
byte_1.byte = byte_2.byte;    /* byte_1.byte = 0x6E */
```

Con la secuencia de código anterior conseguimos reflejar los bits de un byte.

5.4. Alias para los nombres de tipo

Para hacer que un identificador sea considerado de un nuevo tipo, tenemos que emplear la palabra clave *typedef*.
Ejemplo

```
typedef float REAL;
...
REAL x;
```

Con la definición de REAL como sinónimo de *float*, podemos declarar variables de tipo REAL. *Typedef* puede actuar sobre cualquier tipo fundamental o derivado.

5.5. Conversiones de tipo implícitas y explícitas(casting)

Cuando en una expresión se mezclan variables de distintos tipos tienen lugar *conversiones implícitas de tipo*. Por ejemplo, para poder sumar dos variables hace falta que ambas sean del mismo tipo. Si una es *int* y otra *float*, la primera se convierte a *float* (es decir, la variable del tipo de menor rango se convierte al tipo de mayor rango), antes de realizar la operación. A esta conversión automática e implícita de tipo (el

programador no necesita intervenir, aunque sí conocer sus reglas), se le denomina *promoción*, pues la variable de menor rango es promocionada al rango de la otra.

Así pues, cuando dos tipos diferentes de constantes y/o variables aparecen en una misma expresión relacionadas por un operador, el compilador convierte los dos operandos al mismo tipo de acuerdo con los rangos, que de mayor a menor se ordenan del siguiente modo:

long double > double > float > unsigned long > long > unsigned int > int > char

Otra clase de conversión implícita tiene lugar cuando el resultado de una expresión es asignado a una variable, pues dicho resultado se convierte al tipo de la variable (en este caso, ésta puede ser de menor rango que la expresión, por lo que esta conversión puede perder información y ser peligrosa). Por ejemplo, si *i* y *j* son variables enteras y *x* es double,

```
x = i*j – j + 1;
```

En C existe la posibilidad de realizar conversiones explícitas de tipo (llamadas *casting*, en la literatura inglesa). El casting es pues una conversión de tipo, forzada por el programador. Para ello basta preceder la constante, variable o expresión que se desea convertir por el tipo al que se desea convertir, encerrado entre paréntesis. En el siguiente ejemplo,

```
k = (int) 1.7 + (int) masa;
```

la variable *masa* es convertida a tipo *int*, y la constante *1.7* (que es de tipo *double*) también.

6. EXPRESIONES Y OPERADORES

En una expresión van a tomar parte variables, constantes y operadores. Los operadores van a fijar la relación entre las variables y las constantes a la hora de evaluar la expresión. Los paréntesis también pueden formar parte de una expresión y se emplean para modificar la precedencia de los operadores.

6.1. Operadores aritméticos

Hay 5 operadores aritméticos:

+ *Suma*

- *Resta*

* *Multiplicación*

/ *División*. La división de números enteros produce un truncamiento del cociente (por ejemplo, 3/2 = 1).

% *Resto* de una división entera

Las expresiones aritméticas se evalúan de izquierda a derecha. Si en una expresión aritmética intervienen variables o constantes de diferentes tipos, el tipo de resultado va a coincidir con el tipo mayor que aparezca en la expresión. (Por ejemplo, si multiplicamos una variable *float* por una variable *int*, el resultado será *float*).

La suma y la diferencia tienen una representación simplificada mediante los operadores ++ y --, en el caso de sumar o restar la unidad a una variable:

Ejemplo:

```
int x;
...
++x;/* Equivale a x = x+1. Preincremento. */
x++;/* Equivale a x = x+1. Postincremento. */
--x;/* Equivale a x = x-1. Predecremento. */
x--;/* Equivale a x = x-1. Predecremento. */
```

La diferencia entre la posición prefija y la posición sufija de los operadores anteriores queda patente en el ejemplo siguiente:

Ejemplo:

```
int x;
...
x= 5;
printf ("%d\n",++x);/* Incrementa x en 1, por lo que imprime 6. */
x=5;
printf ("%d\n",x++);/* Imprime 5 e incrementa x en 1.*/
```

6.2. Operadores de relación y lógicos

Los operadores de relación y los operadores lógicos se emplean para formar expresiones booleanas. Una expresión booleana sólo puede tomar dos valores: VERDADERO o FALSO. En lenguaje C, se considera que una expresión equivale a una expresión booleana FALSA, cuando al evaluarla su resultado es 0. Si el resultado de una expresión es distinto de 0, se considera que tiene un valor lógico de VERDAD.

Los operadores de relación son:

>	*Mayor*
>=	*Mayor o igual*
<	*Menor*
<=	*Menor o igual*
==	*Igual* (hay que indicar que el operador de asignación es =, y el de comparación ==)
!=	*Distinto*

Los operadores lógicos son:

&&	And lógica.
||	Or lógica.
!	Negación lógica.

Las tablas de verdad de los operadores lógicos son:

Exp_1	Exp_2	$(Exp_1$ && $Exp_2)$	$(Exp_1$ || $Exp_2)$	$!Exp_1$
VERDAD	VERDAD	VERDAD	VERDAD	FALSO
VERDAD	FALSO	FALSO	VERDAD	FALSO
FALSO	VERDAD	FALSO	VERDAD	VERDAD
FALSO	FALSO	FALSO	FALSO	VERDAD

Tabla 1: Tablas de verdad de los operadores lógicos

6.3. Operadores para el manejo de bits

El lenguaje C también implementa operadores para manipular los bits de las variables o constantes enteras.
Los operadores para el manejo de bits son:

- **&** And a nivel de bits. Por ejemplo, 1100 & 0101 => 0100
- **|** Or a nivel de bits. Por ejemplo, 1100 | 0101 => 1101
- **^** Or exclusiva (XOR) a nivel de bits. Por ejemplo, 1100 ^ 0101 => 1001
- **~** Negación a nivel de bits o complemento a 1. Por ejemplo, ~1100 => 0011
- **<<** Desplazamiento hacia la izquierda un número de veces.
  Por ejemplo, 0110 << 1 => 1100.
- **>>** Desplazamiento hacia la derecha haciendo una extensión del signo.
  Por ejemplo, 0110 >> 1 => 0011; 1011 >> 1 => 1101

Hay que tener cuidado de no confundir las operaciones a nivel de bit con las operaciones lógicas.

6.4. Expresiones abreviadas

A continuación mostramos una relación de expresiones abreviadas y sus equivalentes:

Expresión abreviada	Expresión equivalente
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x &= y	x = x & y
x |= y	x = x | y
x ^= y	x = x ^ y
x <<= y	x = x << y
x >>= y	x = x >> y

Tabla 2: Expresiones abreviadas.

6.5. Precedencia y asociatividad de operadores

El resultado de una expresión depende del orden en que se ejecutan las operaciones. El siguiente ejemplo ilustra claramente la importancia del orden. Considérese la expresión:

3 + 4 * 2

Si se realiza primero la suma (3+4) y después el producto (7*2), el resultado es 14; si se realiza primero el producto (4*2) y luego la suma (3+8), el resultado es 11. Con objeto de que el resultado de cada expresión quede claro e inequívoco, es necesario definir las reglas que definen el orden con el que se ejecutan las expresiones de C. Existe dos tipos de reglas para determinar este orden de evaluación: las reglas de *precedencia* y de *asociatividad.* Además, el orden de evaluación puede modificarse por medio de paréntesis, pues *siempre se realizan primero las operaciones encerradas en los paréntesis más interiores*. Los distintos operadores de C se ordenan según su distinta *precedencia* o prioridad; para operadores de la misma precedencia o prioridad, en algunos el orden de ejecución es de izquierda a derecha, y otros de derecha a izquierda (se dice que *se asocian* de izda a dcha, o de dcha a izda). A este orden se le llama *asociatividad*.

En la Tabla 3 se muestra la precedencia –disminuyendo de arriba a abajo– y la asociatividad de los operadores de C.

OPERADORES	**ASOCIATIVIDAD**

() [] -> .	izda a dcha
++ -- ! sizeof (tipo) +(unario) -(unario) *(indir.) &(dirección)	dcha a izda
* / %	izda a dcha
+ -	izda a dcha
<< >>	izda a dcha
< <= > >=	izda a dcha
== !=	izda a dcha
&	izda a dcha
^	izda a dcha
|	izda a dcha
&&	izda a dcha
||	izda a dcha
?:	dcha a izda
= += -= *= /= %= &= ^= |= <<= >>=	dcha a izda
, (operador coma)	izda a dcha

Tabla 3: Precedencia y asociatividad de operadores.

7. SENTENCIAS DE CONTROL DE FLUJO

7.1. Proposiciones y bloques

Una proposición es una expresión seguida de punto y coma (;). Una proposición compuesta o bloque es un conjunto de declaraciones y proposiciones agrupadas entre llaves ({}).

Ejemplo:

```
{/* Bloque. */
        int x; /* Declaraciones. */

        x = 20; /* Proposiciones. */
        x += 10;
        printf ("%d\n",x);
}
```

7.2. If-else

Sintaxis:

```
if (expresión)
        proposición1;
else
        proposición2;
```

Si *(expresión)* es VERDADERA (distinta de cero) se ejecuta proposición$_1$; en caso contrario, se ejecuta proposición$_2$. La expresión una vez procesada debe devolver un valor cero como FALSO o distinto de cero como VERDADERO.

Ejemplo:

```
int x, y, z;
...
/* Cálculo del máximo de 2 números. */
if (x > y)
        z = x;
else
        z = y;
```

Tanto proposición$_1$ como proposición$_2$ pueden ser bloques de código.
El operador ternario ?: se puede utilizar de forma similar a la proposición if-else.

(expresión) ? proposición$_1$: proposición$_2$;

Ejemplo: Macro para calcular el máximo de dos números:

```
#define max(x, y) ((x) > (y)) ? (x): (y))
...
int z;
z = max (10+3, 11);/* z = 13.*/
```

Cuando se quieren anidar series de comprobaciones se suele utilizar ***if*** junto con la cláusula ***else if*** de sintaxis:

if (expresión$_1$)
        proposición$_1$;
else if (expresión$_2$)
        proposición$_2$;
...
else if (expresión$_{n-1}$)
        proposición$_{n-1}$;
else
        proposición$_n$;

La Tabla 4 muestra cuál es la proposición que se ejecuta en función de los valores booleanos de la expresión.

expresión$_1$	**expresión$_2$**	**...**	**expresión$_{n-1}$**	**Proposición que se ejecuta.**
VERDAD	X			proposición$_1$
FALSO	VERDAD	X	X	proposición$_2$
...	...	...	...	...
FALSO	FALSO	FALSO	VERDAD	proposición$_{n-1}$
FALSO	FALSO	FALSO	FALSO	proposición$_n$

Tabla 4: Ejecución de la proposición *else-if*.

7.3. Switch

La proposición *switch* es una decisión múltiple que prueba si una expresión coincide con alguno de entre un número de valores constantes enteros y traslada el control adecuadamente. Sintaxis:

```
switch (expresión)
{
        case exp_const1:proposiciones1;
        case exp_const2:proposiciones2;
        ...
        case exp_constn: proposicionesn;
        default: proposiciones;
}
```

Ejemplo:

```
int mes, día_mes;
...
printf ("Mes: ");
scanf ("%d",&mes);
switch (mes)
{
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
                dias_mes = 31;
                break;
        case 4:
        case 6:
        case 9:
        case 11:
                dias_mes = 30;
                break;
        case 2:
                dias_mes = 28;
                break;
        default:
                printf ("[%d] mes erróneo.\n", mes);
}
```

La proposición *break* provoca una salida anticipada del bucle que la contiene, *switch* en este caso.

7.4. Bucles for

Sintaxis:

for (inicialización**;** expresión**;** progresión**)**
proposición;

Mientras *expresión* sea VERDADERA, se estará ejecutará *proposición*. *Inicialización* es una expresión, o conjunto de expresiones, para inicializar las variables de control que intervienen en *expresión*, y *progresión* es una expresión o conjunto de expresiones que indica cómo evolucionan las variables de control. El ciclo que tiene lugar es como sigue (cuidado con el orden):

1º Se ejecuta *inicialización*
2º Se verifica *expresión* y continua si es cierta
3º Se ejecuta *proposición*
4º Se ejecuta *progresión*
5º Se vuelve al punto 2º

Ejemplo:

```
int i;
...
for (i=0; i < 10; i++)
        printf ("i = %d\n", i);
```

7.5. Bucles while

Sintaxis:

While (expresión**)**
proposición**;**

Proposición se estará ejecutará mientras *expresión* sea VERDADERA.

Ejemplo:

```
int i = 0;
...
while (i < 10)
        printf ("i = %d\n", i++);
```

7.6. Bucles do-while

Sintaxis:

do {
proposiciones;
}**while** (expresión);

Proposición se estará ejecutando mientras *expresión* sea VERDADERA. La primera vez siempre se ejecuta.

Ejemplo:

```
int i = 0;
```

```
...
do {
        printf ("i = %d\n", i++)
}while (i < 10);
```

7.7. Break y continue

La proposición *break* proporciona una salida anticipada de un bucle *for*, *while* o *do*, tal como vimos en la sentencia *switch*.

Ejemplo:

```
int i = 0;
...
for (i = 0; i < 100; i++);
{
        printf ("i = %d\n", i);
        if (i == 20)
                break; /* Este break provoca que el bucle solo llegue hasta 20. */
}
```

La proposición *continue* provoca que se inicie la siguiente iteración del bucle *for*, *while* o *do* que la contiene.

Ejemplo:

```
int i = 0;
...
/* Bucle para imprimir los números múltiplos de 5 menores de 100. */
while (i < 100)
{
        if (i++%5)
                continue;
        printf ("%d\n", i-1);
}
```

8. FUNCIONES

Las funciones van a permitir agrupar bajo un identificador una serie de proposiciones concebidas para realizar alguna tarea específica. La organización de un programa grande en funciones más sencillas hará que el programa sea estructurado además de fácil de depurar y mantener.

Definición de una función según estándar ANSI:

```
tipo nombre_función (declaración_parámetros)
{
        variables_locales;
        proposiciones;
        return (expresión);
}
```

Kernighan y Ritchie, en la primera edición de El lenguaje de programación C, utilizan la siguiente sintaxis para la definición de funciones:

```
tipo nombre_función (parámetros)
declaración_parámetros;
{
        variables_locales;
        proposiciones;
        return (expresión);
}
```

Ejemplo:

```
int factorial (n)
int n;
{
        int i, factorial = 1;

        for (i = 1; i <=n; i++)
                factorial *= i;
        return factorial;
}
```

El mismo ejemplo según el estándar ANSI sería:

```
int factorial (int n)
{
        int i, factorial = 1;

        for (i = 1; i <=n; i++)
                factorial *= i;
        return factorial;
}
```

Una función puede ser de cualquier tipo, excepto tipo función o array. Es decir, una función no puede devolver otra función, ni tampoco un array. Si no se especifica nada, el tipo de una función es *int*.

Las variables locales y los parámetros de la función se reservan en la pila de usuario del programa, por lo que al entrar en la función se reserva espacio para ellos, pero al salir de la función desaparecen. Este tipo de almacenamiento se conoce como *automático*, en contraposición al almacenamiento *estático*. Si una variable local va precedida del calificador **static**, su almacenamiento será estático y ocupará la zona de memoria reservada a las variables globales; además, esa variable existirá durante todo el tiempo de ejecución del programa.

Existen cuatro especificadores de clase de almacenamiento admitidos en C. Estos especificadores indican al compilador cómo debe almacenar la variable que le sigue. El especificador de almacenamiento precede al resto de la declaración de una variable. Los cuatro tipos son los siguientes:

- **extern**: indica al compilador que los nombres y tipos de variables que siguen ya han sido declarados en alguna otra parte como variables globales. Así se pueden enlazar juntos distintos módulos de un gran programa compilados por separado.
- **static**: son variables permanentes, como ya se ha comentado anteriormente. Tienen efectos diferentes cuando son locales que cuando son globales.
- **register**: intenta mantener el valor de la variable declarada con este especificador en un registro de la CPU en lugar de en memoria. Las variables *register* no tienen direcciones (no se pueden apuntar con punteros). No se permiten variables globales *register*.
- **auto**: puede ser usada para declarar variables locales. Sin embargo, como todas las variables que no son globales son, por defecto, asumidas como *auto*, esta palabra casi nunca se usa.

8.1. Pasando parámetros a funciones

Es importante tener en cuenta que los parámetros de una función sólo se pueden modificar a nivel local, es decir, en lenguaje C los parámetros siempre se pasan por valor, es decir, la función tiene una copia de las variables que se le pasan.

Ejemplo:

```
int f(x)
int x;
{
        x = 20;
}
main ()
{
        int x = 10;
        f (x);
        printf ("%d\n", x );    /* Esto va a imprimir el valor 10, ya que la modificación de
x                                                       dentro de la función es local a ella. */
      }
```

Para modificar un parámetro de forma permanente, hay que trabajar con un puntero al mismo. Así, el programa anterior se podría escribir como sigue.

Ejemplo:

```
int f(x)
int *x;
{
*x = 20;
}
        main ()
{
        int x = 10;
        f (&x);
```

```
        printf ("%d\n", x);     /* Esto va a imprimir el valor 20, ya que la modificación de
x dentro                                de la función es a través de un puntero. */
  }
```

Pero también se puede hacer que la función devuelva esa variable. Ejemplo:

```
int f()
{
return 20;      /* La función devuelve 20 */
}
        main ()
{
        int x = 10;
        x=f();
                printf ("%d\n", x);     /* Esto va a imprimir el valor 20, devuelto por la
función */
        }
```

El paso de arrays como parámetros de funciones siempre se realiza pasando el nombre del puntero. Las estructuras se pueden pasar por valor o mediante su puntero. Si pasamos una estructura por valor, las modificaciones de sus campos serán locales mientras que pasándolas por medio de punteros, las modificaciones serán permanentes.

El hecho de que los parámetros pasados por valor sólo sufran modificaciones a nivel local, se debe a que el parámetro es una copia en la pila de usuario de la variable a que se refiere. Así, las modificaciones se hacen sobre la copia de la variable que hay en la pila y no sobre la propia variable.

Como ejemplo, vamos a ver un programa donde se resumen los conceptos de matrices, punteros, estructuras y funciones. El listado siguiente corresponde a un programa para multiplicar matrices. Se encuentra estructurado en una librería y un programa principal, y se incluye el *Makefile* para su compilación.

Programa 3 Multiplicación de matrices (matrices.c, libmatrices.h, libmatrices.c, Makefile)

fichero "matrices.c"

```
/***
PROGRAMA PRINCIPAL; matrices.c
DESCRIPCIÓN: Programa para multiplicar matrices de números reales.
***/

#include <stdio.h>     /* Librerías habituales */
#include <stdlib.h>
#include "libmatrices.h"        /* Incluye la librería de operaciones con matrices */


/***        Funciones       ***/
```

```
/*
FUNCIÓN: imprimir_matriz  DESCRIPCIÓN: Presenta una matriz por pantalla.
*/

imprimir_matriz (struct matriz m)
{
		int i, j;

		for (i = 0; i < m.filas; i++)
		{
			for (j = 0; j < m.columnas; j++)
				printf ("%g ", m.coef [i][j]);
			printf ("\n");
		}
}


/***	Función principal	***/

main ()
{
		struct matriz a, b, c;

		/* Pide las matrices de entrada*/
		printf("\n-- Primera matriz:\n");
		a = leer_matriz ();
		printf("\n-- Segunda matriz:\n");
		b = leer_matriz ();

		/* Multiplica */
		if (a.columnas!=b.filas)
			error("Imposible multiplicar ambas matrices.\n");

		c = multiplicar_matrices (a, b);

		/* Presenta el resultado en pantalla */
		printf("\n-- Matriz resultado:\n");
		imprimir_matriz (c);

		/* Libera la memoria dinámica ocupada por las matrices */
		LiberaMatriz(a);
		LiberaMatriz(b);
		LiberaMatriz(c);
```

fichero "libmatrices.h"

```
/***
LIBRERIA; libmatrices.h
DESCRIPCIÓN: archivo de cabecera de la librería libmatrices.c
***/

/*** Declaración de tipos ***/

/* Definición de matriz. */
struct matriz
{
        int filas, columnas;
        float **coef;
};


/*** Declaración de funciones de la librería ***/

/* Reserva  dinámica de memoria para los coeficientes de una matriz */
float **reservar_coeficientes (int filas, int columnas);

/* Lee una matriz. Filas, columnas y coeficientes. Devuelve la matriz leída */
struct matriz leer_matriz ();

/* Recibe como parámetros dos matrices y devuelve el resultado de su producto */
struct matriz multiplicar_matrices (struct matriz a, struct matriz b);

/* Libera la zona de memoria de los coeficientes de una matriz */
void LiberaMatriz(struct matriz m);

/* Presenta el último error producido */
void error (char *str);
```

fichero "libmatrices.c"

```
/***
LIBRERIA; libmatrices.c
DESCRIPCIÓN: librería con operaciones sobre matrices
***/

#include <stdio.h>
#include <stdlib.h>
#include "libmatrices.h"   /* Incluye su propio archivo de cabecera */

/***
        FUNCIÓN: reservar_coeficientes
        DESCRIPCIÓN: Reserva  dinámica de memoria para los coeficientes de una matriz.
***/
float **reservar_coeficientes (int filas, int columnas)
{
        float **coef;
        int i;

        if ((coef = (float **) malloc (filas * sizeof (float *))) == NULL)
                error (" Error asignando memoria");
        for (i = 0; i < filas; i++)
                if ((coef [i] = (float *) malloc (columnas * sizeof (float))) == NULL)
                        error (" Error asignando memoria");
        return (coef);
}


/***
        FUNCIÓN : leer_matriz
        DESCRIPCIÓN: Lee una matriz. Filas, columnas y coeficientes. Devuelve la matriz leída.
***/
struct matriz leer_matriz ()
{
        struct matriz m;
        int i, j;

        /* Pide al usuario las dimensiones de la matriz */
        printf ("Dimensiones de la matriz (Ej: 2 3  -> indica una matriz 2*3):\n");
        scanf ("%d %d", &m.filas, &m.columnas);
        /* Reserva la zona de memoria para los coeficientes */
        m.coef=reservar_coeficientes(m.filas, m.columnas);
        /* Pide los elementos de la matriz */
        printf ("Coeficientes de la matriz (coeficiente1,1 <ENTER> coeficiente1,2 <ENTER>...):\n");
        for (i = 0; i < m.filas; i++)
                for (j = 0; j < m.columnas; j++)
                        scanf ("%f", &m.coef [i][j]);
        return (m);
```

```
}


/***
        FUNCIÓN: multiplicar_matriz.
        DESCRIPCIÓN: Recibe como parámetros dos matrices y devuelve el resultado de su producto.
***/
struct matriz multiplicar_matrices (struct matriz a, struct matriz b)
{
        struct matriz c;
        int i, j, k;

        if (a.columnas != b.filas)
                error ("Las matrices no se pueden multiplicar");
        c.filas = a.filas;
        c.columnas = b.columnas;
        c.coef = reservar_coeficientes (c.filas, c.columnas);

        for (i = 0; i < c.filas; i++)
                for (j = 0; j < c.columnas; j++)
                {
                        c.coef[i][j] = 0;
                        for (k = 0; k < a.columnas; k++)
                                c.coef[i][j] += a.coef[i][k]*b.coef[k][j];
                }
        return (c);
}


/***
        FUNCIÓN: LiberaMatriz
        DESCRIPCIÓN: Libera la zona de memoria de los coeficientes de una matriz.
***/
void LiberaMatriz(struct matriz m)
{
        int i,j;

        /* Libera las columnas */
        for (i = 0; i < m.filas; i++)
                free(m.coef[i]);
        /* Libera el apuntador a las filas */
        free(m.coef);
}


/***
        FUNCIÓN: error
        DESCRIPCIÓN: Presenta el último error producido.
***/
void error (char *str)
```

```
{
        printf ("\nERROR: %s.\n", str);
        exit (-1);                          /* Sale del programa */
}
```

fichero "Makefile"

```
all: matrices

matrices: matrices.c libmatrices.o
        gcc -o matrices matrices.c libmatrices.o

libmatrices.o: libmatrices.h libmatrices.c
        gcc -c libmatrices.c
```

8.2. La función main

Como ya se ha comentado, la función main debe existir en todo programa C y sólo en uno de los ficheros fuente que se compilan para crear un programa. Además es la función por la que comienza la ejecución del programa, por ello, es en ella donde se pueden obtener las opciones pasadas en la línea de comandos.

Cuando se ejecuta un programa desde la linea de comandos tecleando su nombre, existe la posibilidad de pasarle algunos datos, tecleándolos a continuación en la misma línea. Por ejemplo, se le puede pasar algún valor numérico o los nombres de algunos ficheros en los que tiene que leer o escribir información. Esto se consigue por medio de argumentos que se pasan a la función **main()**, como se hace con otras funciones.

Así pues, a la función **main()** se le pueden pasar argumentos y también puede tener valor de retorno. El primero de los argumentos de **main()** es una variable *int* que contiene el número de palabras (separadas por espacios) de la linea de comandos incluyendo el nombre del programa. El segundo argumento es un *vector de punteros a carácter* que contiene las direcciones de la primera letra o carácter de dichas palabras. A continuación se presenta un ejemplo:

```
int main(int argc, char *argv[])
{
        int cont;
        for (cont=0; cont<argc; cont++)
        printf("El argumento %d es: %s\n", cont, argv[cont]);
        printf("\n");
        return 0;
}
```

9. E/S POR CONSOLA

Las funciones de E/S por consola controlan la entrada de datos por el teclado y la salida a través de pantalla.

Las funciones principales van a ser las siguientes :

9.1. Lectura y escritura de caracteres

getchar()

Lee un carácter del teclado. Espera hasta que se pulsa una tecla y entonces devuelve su valor. El prototipo es : *int getchar(void) ;*

putchar()

Imprime un carácter en la pantalla. El prototipo es : *int putchar(int c) ;*

9.2. Lectura y escritura de cadenas

gets()

Lee una cadena de caracteres introducida por el teclado y la coloca en la dirección apuntada por su argumento. Como detalle, el gets no pasa el carácter de fin de línea '\n' por lo que se queda en el buffer de lectura.

El prototipo es : char *gets(char *cad) ;

puts()

Escribe una cadena en la pantalla. Escribe su argumento de tipo cadena en la pantalla seguido de un carácter de salto de línea.

El prototipo es : int puts(const char *cad) ;

La cadena se define como *const* porque la fucnión no la modifica. Otra utilidad de *const* es para definir una constante, por ejemplo:

const double pi=3.1416;

9.3. E/S por consola con formato

scanf()

Devuelve el número de datos a los que se ha asignado un valor con éxito. Es la rutina de entrada por consola. Actúa como si fuera la inversa de **printf()**.

El prototipo es : int scanf(const char *cadena_de_control,...) ;

Los especificadores de formato de entrada van precedidos por el signo % e indican a **scanf()** qué tipo de datos se va a leer a continuación. Coinciden con los de printf().

printf()

Escribe datos en la consola. Devuelve el número de caracteres escritos o un valor negativo si se ha producido un error.

El prototipo es : int printf(const char *cadena_de_control,...) ;

La cadena_de_control está formada por dos tipos de elementos. El primer tipo está compuesto por los caracteres que se mostrarán en la pantalla. El segundo tipo contiene especificadores de formato que definen la forma en que se muestran los argumentos posteriores

Un especificador de formato empieza con un signo de porcentaje (%) y sigue con el código del formato de la variable que se quiere imprimir.

Ejemplo : %c => se utiliza para imprimir un sólo carácter.
%d => se utiliza para indicar un número entero con signo.

A continuación se citan las conversiones más típicas.

Carácter	Conversión del tipo de argumento
d, i	entero con signo
o	octal
x, X	hexadecimal
u	entero sin signo
c	carácter
s	cadena de caracteres (terminada en '\0')
f	en punto fijo
e, E	notación científica
p	imprime un apuntador (la posición de memoria a la que apunta)
%	cuando se quiere imprimir un '%' vale con poner dos seguidos "%%"

Tabla 5: Carácteres de control % para printf() y scanf().

Otros caractéres especiales son:

Carácter	Significado
'\a'	Alarma (pitido)
'\0'	Indicador de fin de línea
'\n'	Salto de línea
'\t'	Tabulador

Tabla 6: Caractéres especiales.

Ejemplo de scanf() y printf() :

```
#include <stdio.h>
void main(void)
{
        int x ;
        printf("Introduce un número :") ;
        scanf("%d",&x) ;
        printf("El número introducido es :%d",x) ;
}
```

El archivo de cabecera para las funciones anteriormente expuestas es *stdio.h*, es decir, al principio del programa es necesaria una linea como la siguiente:

#include <stdio.h>

10. E/S POR ARCHIVOS

Las principales funciones de entrada y salida definidas por el estándar ANSI son las que se van a mostrar a continuación, y todas ellas requieren que se incluya el archivo de cabecera *stdio.h* en cualquier programa en que se vayan a utilizar.

fopen()
Abre un *stream* (fichero) para que pueda ser utilizada y vincula un archivo con la misma. Después, devuelve el puntero al archivo asociado con ese archivo.

El prototipo es : FILE *fopen(const char *nombre, const char *modo) ;

fclose()
Cierra un *stream* que haya sido abierta con una llamada a **fopen()**. Cierra un archivo.

El prototipo es : int fclose(FILE *fp) ;

putc(), fputc()
Escribe caracteres en un archivo que haya sido previamente abierto para operaciones de escritura con **fopen()**.

El prototipo es : int putc(int c, FILE *fp) ;

getc(), fgetc()
Lee caracteres de un archivo abierto en modo lectura con **fopen()**.

El prototipo es : int getc(FILE *fp) ;

Las funciones mostradas anteriormente constituyen el conjunto mínimo de rutinas de tratamiento de archivos.

feof()
Determina cuándo se ha alcanzado el final del archivo. Devuelve cierto si se llega al final del archivo.

El prototipo es : int feof(FILE *fp) ;

fputs()
Escribe cadenas de caracteres sobre archivos de disco. Si se produce algún error, devuelve EOF.

El prototipo es : int fputs(const char *cad, FILE *fp) ;

fgets()
Lee una cadena de la secuencia especificada hasta que se llega a un carácter de salto de línea o se hayan leído *longitud-1* caracteres. La función devuelve *cad* si se ha ejecutado correctamente y un puntero nulo si se produce algún error.

El prototipo es : char *fgets(char *cad, int longitud, FILE *fp) ;

rewind()
Reinicializa el indicador de posición al principio del archivo especificado por su argumento.

El prototipo es : void rewind(FILE *fp) ;

ferror()
Determina si se ha producido un error en una operación sobre un archivo. En este caso, devuelve cierto.
El prototipo es : int ferror(FILE *fp) ;

remove()
Elimina el archivo especificado.
El prototipo es : int remove(const char *nombre) ;

fflush()
Vacía el contenido de una secuencia de salida. Escribe todos los datos del buffer en el archivo asociado con *fp.*
El prototipo es : int fflush(FILE *fp) ;

fread()
Permite la lectura de tipos de datos que ocupen más de 1 byte.
El prototipo es :
size_t fread(void *buffer, size_t num_bytes, size_t cuenta, FILE *fp) ;

fwrite()
Permite la escritura de tipos de datos que ocupen más de 1 byte.
El prototipo es :
size_t fwrite(const void *buffer, size_t num_bytes, size_t cuenta, FILE *fp) ;

fseek()
Busca un byte específico de un archivo.
El prototipo es : int fseek(FILE *fp, long numbytes, int origen) ;

fprintf() y fscanf()
Hacen lo mismo en archivos que **printf()** y **scanf()** en la consola.
Sus prototipos son :
int fprintf(FILE *fp, const char *cadena_de_control,...) ;
int fscanf(FILE *fp, const char *cadena_de_control,...) ;

Ejemplo:

```
/* Este programa escribe un float en el archivo prueba */
#include <stdio.h>
#include <stdlib.h>

void main()
{
        FILE *fp;
 float  f=12.23;

         if ((fp=fopen("prueba", "wb"))==NULL) {
              printf("No se puede abrir el fichero.\n");
              exit(1);
 }
 fwrite(&f,sizeof(float),1,fp);
 fclose(fp);
}
```

11. FUNCIONES ESTÁNDAR DE LA LIBRERÍA DE FUNCIONES

Las funciones, tipos y macros de la librería estándar están declaradas en las cabeceras estándar siguientes :

<float.h> <math.h> <stdarg.h> <stdlib.h> <ctype.h> <limits.h>
<stddef.h> <string.h> <errno.h> <locale.h> <signal.h> <stdio.h>
<time.h>

Las cabeceras se incluyen así : #**include** <*cabecera*>

A continuación se va a mostrar un listado de algunas de las principales funciones incluidas en las cabeceras estándar.

<ctype.h> Manejo de caracteres.
<errno.h> Informes de error.
<float.h> Define valores en coma flotante dependientes de la implementación.
<limits.h> Define varios límites dependientes de la implementación.

CHAR_BIT	8	Bits en un carácter.
INT_MAX	+32767	Máximo valor de un entero.
INT_MIN	-32767	Mínimo valor de un entero.

<math.h> Varias definiciones usadas en la biblioteca matemática como: sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), exp(x), log(x), pow(x,y), sqrt(x), fabs(x), fmod(x,y).
<stddef.h> Define algunas constantes de uso común.
<stdio.h> Soporte para E/S de archivos. Explicado en secciones anteriores.
<stdlib.h> Declaraciones variadas.

int atoi(const char *s)	Convierte s a un entero.
int rand(void)	Devuelve un entero pseudo-aleatorio de 0 a RAND_MAX.
void *malloc(size_t size)	Reserva memoria dinámicamente.
void free(void *p)	Libera memoria reservada por *malloc().

<string.h> Soporte para las funciones de cadenas.

char *strcpy(s,ct)	Copia el string ct al string s, incluyendo '\0' ; devuelve s.
char *strcat(s,ct)	Concatena el string ct al final del string s ; devuelve s.
int strcmp(cs,ct)	Compara el string cs con el ct
size_t strlen(cs)	Devuelve la longitud de cs.

<time.h> Soporte para las funciones de tiempo del sistema.

En este manual no se han incluido todas las funciones que se encuentran en la librería estándar. Para más detalle, remitirse a los libros recomendados.

12. MANUAL EN LINEA DE UNIX

Resulta de interés el comando de los sistemas UNIX denominado *man*, que proporciona información acerca de la palabra especificada detrás suyo. Por ejemplo, si se quiere información sobre la función seno, incluída en la librería *math.h*, se indicaría de la siguiente forma:

 # man sin

Saldrá una explicación detallada de la función, y también alusiones a otras funciones de la misma librería.

La información del *man* está dividida en secciones según el tema sobre el que versen. Las más utilizadas son la secciones:

1 - Comandos de usuario
2 - Llamadas al sistema
3 - Biblioteca de funciones C

Si se quiere, por ejemplo, información de la llamada al sistema *open*, que está en la sección 2, se indicaría así:

 # man 2 open

13. Bibliografía

Libros

Brian W. Kernighan y Dennis M. Ritchie, "El lenguaje de programación C", Prentice Hall 2ªedición 1991, ISBN 968-880-205-0
Byron S. Gjottfried, "Programación en C", Mc Graw-Hill, 1991.
K. N. King, "C PROGRAMMING: A Modern Approach", Georgia State University, ISBN 0-393-96945-2

Links con tutoriales de lenguaje C

http://www.cm.cf.ac.uk/Dave/C/CE.html
http://www.cyberdiem.com/vin/learn.html
http://www.lysator.liu.se/c/bwk-tutor.html
http://www.strath.ac.uk/IT/Docs/Ccourse/

INTRODUCCIÓN A LAS LLAMADAS AL SISTEMA OPERATIVO LINUX

Introducción de llamadas al sistema operativo LINUX

ÍNDICE

1.- Introducción

En este documento se van a analizar algunas llamadas al Sistema Operativo LINUX.

En un primer apartado, se hará un repaso de los conceptos generales de LINUX, nombrando los tipos de ficheros que existen, definiendo programas y procesos, los identificadores y los permisos asociados. Estos son conceptos que ya se vieron en las prácticas de *Introducción básica al Sistema Operativo LINUX a nivel de usuario*.

En apartados posteriores, se tratará el formato de las llamadas al sistema y toda la documentación relacionada con estas llamadas. Se clasificarán en *Llamadas de acceso a ficheros, Llamadas de control de procesos, Interrupciones software: señales,* y *Comunicaciones entre procesos*.

2.- Conceptos generales

Un sistema operativo es el programa o conjunto de programas que permiten gestionar los recursos del hardware de un sistema informático como son la memoria, dispositivos de almacenamiento de datos, terminales, etc. Los diferentes programas que se desarrollan en una máquina se basan en las peticiones de servicios a dicho sistema operativo.

LINUX como sistema operativo ofrece servicios para manejar la memoria, la planificación de los procesos, la comunicación entre los mismos, y operaciones de entrada y salida. Los procesos o programas pueden solicitar dichos servicios directamente al **KERNEL** (llamadas al sistema), o indirectamente a través de rutinas de librería. La comodidad de utilizar el lenguaje C viene del hecho de que se puede utilizar cualquier llamada al sistema a través de las bibliotecas de funciones de que se dispone.

Ficheros

Los ficheros en LINUX se organizan en lo que se conoce como **file systems** (sistemas de ficheros), que son agrupaciones de ficheros bien por razones físicas o lógicas. Hay tres tipos de ficheros en LINUX:

- fichero ordinario: conjunto de bytes en un ordenamiento secuencial. Cualquier byte de dicho ordenamiento puede ser leído y escrito. Tan solo se puede modificar el fichero añadiendo o eliminando bytes del final del mismo. Un fichero ordinario se identifica por un número denominado **i-number**. Un i-number es un índice a una tabla de **i-nodos** que se mantiene al comienzo de cada sistema de ficheros. Cada **i-nodo** contiene información del tipo: tipo de fichero, propietario, grupo, permisos, etc.
- directorio: permite utilizar nombres lógicos para referirnos a todos los ficheros. Consiste en una tabla de dos campos: el primer campo contiene el nombre de un fichero, y el segundo campo su correspondiente i-number. Cada registro de dicha

tabla (nombre, i-number) se conoce como **enlace (link)**. Cuando se solicita al sistema operativo el acceso a un fichero referenciándolo por su nombre, éste busca en un directorio para encontrar el i-number. Entonces accede al inodo asociado, donde se encuentra la información asociada al fichero y las direcciones de dónde se encuentra en el disco.

- fichero especial: puede ser de dos clases, FIFO y dispositivo.

FIFO (first-in-first-out queue) es un mecanismo de intercambio de datos entre procesos.

- dispositivo (device): es cualquier dispositivo hardware.

Los ficheros especiales también tienen un i-nodo asociado, pero dicho i-nodo no apunta a ningún bloque en disco donde se contienen datos sino que se hace referencia a una tabla que usa el kernel donde se encuentran unas rutinas llamadas driver. El driver sirve para la comunicación con los terminales. Si se trata de un proceso de entrada de datos desde un dispositivo externo, los datos son pasados a un interface y es el driver el que realiza un procesamiento de éstos y entrega los datos al programa. Si por el contrario, desde el proceso se envían los datos, el driver procesa los datos de salida y los envía al interface.

Programas y procesos

Se entiende por **programa** al conjunto de instrucciones y datos que se encuentran en un fichero normal en el disco. El i-nodo asociado está marcado como ejecutable.

Un programa en ejecución se conoce como **proceso**. LINUX distingue dentro de un proceso tres regiones básicas: texto, datos y pila. Texto es el conjunto de bytes que se interpretan como instrucciones por la CPU. Los datos son tratados por el procedimiento definido en el programa y pila es la zona dinámica de la memoria de un proceso que permite la implementación de rutinas y paso de parámetros. Varios procesos pueden compartir algunas de estas regiones.

Un proceso se puede encontrar en diferentes estados, como listo para ejecutarse (ready to run), ejecutándose (running), o durmiendo (sleeping).

El **system data segment** (segmento de datos del sistema) de un proceso consiste en una serie de información que mantiene el kernel para controlar la ejecución de los procesos. Esta información (process id, parent process id, process group id, current directory, etc) se encuentra repartida en diferentes tablas del sistema.

Identificadores de procesos y grupos

Todos los procesos en LINUX tienen un número positivo que los identifica, el **process-ID (pid)**. Todos los procesos excepto algunos del sistema tienen un proceso padre. El identificador del proceso padre se conoce con el nombre de **parent process-ID (ppid)**.

Si el padre de un proceso muere (termina), éste es adoptado por el proceso **init** (pid=1).

Permisos

Cada usuario del sistema tiene asociado un número positivo llamado **user ID**. Los usuarios pueden estar organizados en grupos, cada grupo tiene una identificación conocida con el nombre de **group ID**.

Cuando un usuario entra en el sistema se leen los ficheros en donde se configuran los usuarios y los grupos y se asocian al usuario las identificaciones de usuario y grupo.

Cada fichero en su i-nodo contiene información del user ID y del group ID del propietario. Así mismo, contiene los conjuntos de permisos que ya se indicaron en la *Introducción básica al Sistema Operativo LINUX a nivel de usuario.* Estos conjuntos de permisos son: usuario, grupo y otros, y dentro de cada uno de ellos, permisos de lectura, escritura y ejecución.

3.- Formato general de las llamadas al sistema

Las llamadas al sistema se pueden usar por medio de funciones en C y todas ellas tienen un formato común, tanto en su documentación como en la secuencia de llamada.

Una llamada al sistema se invoca mediante una función retornando ésta siempre un valor con información acerca del servicio que proporciona o del error que se ha producido en su ejecución. Aunque dicho retorno puede ser ignorado, es recomendable siempre testearlo.

Retornan un valor de -1 como indicación de que se ha producido un error en su ejecución. Además existe una variable externa `errno`, donde se indica un código de información sobre el tipo de error que se ha producido. En el archivo de include `errno.h` hay declaraciones referentes a esta variable.

La variable `errno` no cambia de valor después de una llamada al sistema que retorna con éxito, por tanto es importante tomar dicho valor justo después de la llamada al sistema y únicamente cuando éstas retornan error.

El formato general de documentación de una llamada al sistema es:

```
CREAT(2)
 Crear un fichero nuevo.
SYNOPSIS
 #include <sys/types.h>
 #include <sys/stat.h>
 #include <fcntl.h>

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

```
DESCRIPTION
 Creat crea un fichero regular nuevo, o prepara para volver a
escribir uno ya existente.
 .......
ERRORS
 [ENOSPC] no hay suficiente espacio en el sistema de ficheros.
 [EACCES] no se tiene permiso de acceso.
 ......
RETURN VALUE
 Si no hay error retorna el file descriptor, si hay algún error
retorna un -1 y se guarda en la variable errno la causa del error.
SEE ALSO
 chmod(2), close(2),...
```

Nota: La información incluida en este manual, referente a llamadas al sistema, no pretende ser, ni mucho menos, exhaustiva ni completa. Se recomienda recurrir siempre al manual en línea (man) antes de emplear cualquiera de ellas. Se recuerda que la sección 2 del manual es la dedicada a llamadas al sistema.

4.- Llamadas de acceso a ficheros

En LINUX, los procesos acceden a los ficheros por medio de **file descriptors (descriptores de fichero)**. Un file descriptor es un índice a una tabla de descriptores de fichero **(file descriptor table)**, que mantiene el kernel para el proceso. Asociada a esa entrada en la tabla hay información referente al modo en que se está empleando ese fichero, como puede ser la posición en el fichero a partir de la cual se realizará la próxima operación de lectura/escritura que se ejecute sobre ese descriptor o el i-nodo del fichero.

Generalmente, cada proceso comienza con tres file descriptors, que hacen referencia al terminal del proceso:

STDIN_FILENO (0)	standard input
STDOUT_FILENO (1)	standard output
STDERR_FILENO (2)	standard error

Dentro de las llamadas al sistema para el acceso a ficheros destacan las siguientes: `open, read, write, close, lseek, unlink`.

A continuación se va a analizar con detalle la estructura de cada una de ellas.

4.1.- Abrir un fichero (open)

Permite abrir o crear un fichero para lectura y/o escritura. Esta llamada al sistema crea una entrada en la tabla de descriptores de fichero del proceso, retornando al mismo el descriptor asignado, el cual suele ser el primero que encuentre libre en la tabla.

```
OPEN(2)

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

      int open(const char *pathname, int flags [, mode_t
mode]);



DESCRIPCIÓN
      pathname apunta al nombre de un fichero. La xara4661!
llamada open abre el fichero designado por path y devuelve el
descriptor de fichero asociado.
      flags se utiliza para indicar el modo de apertura del
fichero. Dicho modo se construye combinando los siguientes
valores:
      O_RDONLY: modo lectura.
      O_WRONLY: modo escritura.
      O_RDWR: modo lectura y escritura.
      O_CREAT: si el fichero no existe, lo crea con los
permisos indicados en perms.
      O_EXCL: si está puesto O_CREAT y el fichero existe, la
llamada da error.
      O_APPEND: el fichero se abre y el offset apunta al final
del mismo. Siempre que se escriba en el fichero se hará al
final del mismo.
      mode es opcional y permite especificar los permisos que
se desea que tenga el fichero en caso de que se esté creando.

RETORNO
      Retorna el file descriptor del fichero o -1 si hay
error.
```

Existe una llamada al sistema que produce el mismo efecto que el `open` con los flags: O_RDWR | O_CREAT. Esta llamada es `creat` y su sinopsis es:

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

4.2.- Lectura de un fichero (`read`)

La llamada `read` se utiliza para leer un número dado de bytes de un fichero. La lectura comienza en la posición señalada por el descriptor, y tras ella se incrementa ésta en el número de bytes leídos.

```
READ (2)

SINOPSIS
     #include <unistd.h>

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

DESCRIPCIÓN
     La llamada read lee un número de bytes dado por count
del fichero al que hace referencia el descriptor de fichero
fd y los coloca a partir de la dirección de memoria apuntada
por buf.

RETORNO
     Retorna el número de bytes leídos, 0 si encuentra el
final del fichero y -1 si hay error.
```

4.3.- Escritura de un fichero (`write`)

La llamada `write` se emplea para escribir un número de bytes en un fichero. La escritura comienza en la posición señalada por el descriptor y tras ésta se incrementa la posición en el número de bytes escritos.

La escritura se realiza en una memoria caché del sistema y el kernel se encarga de volcar dicha memoria al fichero en disco o en el dispositivo.

Tiene el siguiente formato:

```
WRITE (2)

SINOPSIS
     #include <unistd.h>

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

DESCRIPCIÓN
     La llamada write escribe un número de bytes dado por
count en el fichero cuyo file descriptor viene dado por fd.
Los bytes a escribir deben encontrarse a partir de la posición
de memoria indicada en buf.

RETORNO
     Retorna el número de bytes escrito o -1 si hay un error.
```

4.4.- Cerrar un fichero (`close`)

La llamada close deshace el enlace entre un descriptor de fichero y su fichero. La entrada en la tabla de descriptores de fichero del proceso queda disponible para volver a ser utilizada. Su descripción es la siguiente:

```
CLOSE (2)

SINOPSIS
     #include <unistd.h>

     int close(int fd);

DESCRIPCIÓN
     Close cierra un fichero.

RETORNO
     Retorna 0 si no hay error y -1 si hay algún error.
```

4.5.- Posicionamiento en un fichero (`lseek`)

La llamada `lseek` establece la posición señalada por un descriptor de fichero, la cual será empleada en la próxima llamada a read/write.

```
LSEEK (2)

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

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

DESCRIPCIÓN
     Establece la posición señalada por el descriptor de
fichero de la siguiente forma:
     . si whence es SEEK_SET, el puntero al fichero apunta
a la dirección de offset.
     . si whence es SEEK_CUR, el puntero al fichero apunta
a la dirección actual más offset.
     . si whence es SEEK_END, el puntero al fichero apunta a
la longitud del fichero más offset.

RETORNO
     Retorna la nueva posición señalada por el descriptor de
fichero si no hay error y -1 si hay algún error.
```

4.6.- Destruyendo entradas en los directorios (`unlink`)

Borra una entrada en la tabla de un directorio.

UNLINK (2)

SINOPSIS

```
#include <unistd.h>

int unlink(const char *pathname);
```

DESCRIPCIÓN

Borra una entrada en la tabla de un directorio. pathname representa un fichero ya existente cuya entrada se quiere borrar. Si se elimina la última entrada existente en algún directorio que hace referencia a un i-nodo concreto el sistema elimina el fichero de ese i-nodo.

RETORNO

Retorna 0 si no hay error y -1 si hay algún error.

Programa ejemplo: Copia de fichero

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

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

        if (argc!=3)
        {
              fprintf(stderr, "Se precisan 2 argumentos\n");
              exit(1);
        }

        fdold=open(argv[1], O_RDONLY);
        if (fdold==-1)
        {
        fprintf(stderr, "No se pudo abrir el fichero %s\n", argv[1]);
        exit(1);
        }

        fdnew=creat(argv[2], 0666);
        if (fdnew==-1)
        {
        fprintf(stderr, "No se pudo crear el fichero %s\n", argv[2]);
        exit(1);
        }

        copy(fdold, fdnew);
        exit(0);
}
```

```
copy(int old, int new)
{
          int cuenta;
          char    buffer[2048];

          while ((cuenta=read(old, buffer, sizeof(buffer)))>0)
                write(new, buffer, cuenta);

}
```

5.- Llamadas de control de procesos

5.1.- Conceptos generales

Un proceso se puede entender en parte como un programa en ejecución. Sus características generales son las siguientes:

- Un proceso consta de código, datos y pila.
- Los procesos existen en una jerarquía de árbol (varios hijos, un solo padre).
- El sistema asigna un identificador de proceso (PID) único al iniciar el proceso.
- El planificador de tareas asigna un tiempo de empleo de la CPU para el proceso según su prioridad.

5.2.- Ejecución de programas (`exec`)

Un programa (fichero ejecutable) es ejecutado cuando un proceso hace una llamada `exec` al sistema. El kernel sustituye los segmentos de texto y de datos del proceso que realiza la llamada por los del fichero ejecutable que se le pasa como parámetro en la llamada. El proceso continúa su ejecución en la primera línea del programa ejecutado. Es decir, el proceso sustituye el programa que está ejecutando por otro. Una vez completada con éxito la llamada `exec()` el código del antiguo programa deja de ejecutarse (desaparece ya que es sustituido por el del programa ejecutado) y se pasa al nuevo código de programa. Sin embargo el proceso sigue siendo el mismo, es decir, tiene los mismos identificadores de proceso (PID), de proceso padre (PPID) y de grupo de procesos (PGID), la misma tabla de descriptores de ficheros, mantiene el directorio actual, etc.

La llamada `exec` tiene variantes. Generalmente una de ellas es una llamada al sistema y el resto son funciones de biblioteca que permiten pasar los parámetros de forma más cómoda pero que internamente emplean la llamada al sistema

EXEC (2)

SINOPSIS

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

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

DESCRIPCIÓN

Las 6 funciones tienen la misma funcionalidad de ejecutar un fichero ejecutable.

La diferencia entre las 6 son la forma de pasar los parámetros de entrada.

Los nombres de estas funciones están compuestos por exec y uns serie de letras que tienen el siguiente significado:

- l : Los argumentos para el programa (cadenas de texto) se incluyen uno a uno en los argumentos de la función, indicando el último mediante el puntero 0.
- v : Los argumentos para el programa se pasan mediante un array de punteros a las cadenas de texto que son los argumentos. El último elemento del array debe ser el puntero 0.
- e : Permite pasar las variables globales de entorno que deben definirse para la ejecución del nuevo programa.
- p : La búsqueda del programa se hará en todos los directorios contenidos en la variable de entorno PATH.

RETORNO

Retornan -1 si hay error. Si se ejecuta con éxito no devuelve ningún valor dado que el código de programa se sustituye por el del nuevo programa y se eliminan todas las variables del programa.

Argumentos de un programa

Cuando ejecutamos un programa desde una Shell podemos especificar parámetros a los que queremos que éste tenga acceso. ej:

```
%> cp fichero1 fichero2
```

[cp es el nombre del programa, fichero1 y fichero2 son los parámetros o argumentos]

Un programa escrito en C puede estar preparado para acceder a estos argumentos. En la declaración de la función `main()` se pueden incluir varias variables:

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

argc es un entero que cuenta el número de argumentos que posee el programa

argv es una array de punteros, los cuales hacen referencia a las cadenas de texto donde están los argumentos. El primer argumento es el nombre del ejecutable (argv[0]), es decir, el nombre del programa cuenta entre los argumentos, por lo que argc valdrá al menos 1.

environ es análogo a argv pero contiene las variables de entorno

5.3.- Crear un proceso (`fork`)

Los procesos se crean a través de la llamada al sistema `fork`. Cuando se realiza dicha llamada, el kernel duplica el entorno de ejecución del proceso que llama, dando como resultado dos procesos.

El proceso original que hace la llamada se conoce como *proceso padre,* mientras que el nuevo proceso resultado de la llamada se conoce como *proceso hijo.*

La diferencia en los segmentos de datos de ambos procesos es que la llamada `fork` retorna un 0 al proceso hijo, y un entero que representa el PID del hijo al proceso padre. Si la llamada fracasa, no se crea el proceso hijo y se devuelve -1 al proceso padre.

Una vez ejecutada con éxito la llamada `fork` y devueltos los valores de retorno ambos procesos continúan su ejecución a partir de la siguiente instrucción al fork.

```
FORK (2)

SINOPSIS
      #include <unistd.h>

      pid_t fork(void);

DESCRIPCIÓN
      fork causa la creación de un nuevo proceso copia (casi
exacta) del proceso padre.

RETORNO
      si la llamada tiene éxito retorna:
      0         al proceso hijo
      pid del hijo   al proceso padre
      si fracasa devuelve -1
```

5.3.1.- Programa ejemplo de utilización de `fork`

```
/* fork.c - Ejecución conjunta de procesos padre e hijo */
#include <stdio.h>
#include <unistd.h>
main ()
    {
    pid_t  pid;
    printf ("Ejemplo de fork.\n");
    printf ("Inicio del proceso padre. PID=%d\n", getpid ());

    pid=fork();
    if (pid == 0)
        {       /* Proceso hijo */
        printf ("Proceso hijo. PID=%d, PPID=%d\n", getpid (),
              getppid ());
        sleep (1);
        }
    else
        {       /* Proceso padre */
        printf ("Proceso padre. PID=%d\n", getpid ());
        sleep (1);
        }
    printf ("Fin del proceso %d\n", getpid ());
    exit (0);
    }
```

5.4.- Espera de la terminación de un proceso (`wait`)

Cuando un proceso termina se envía un *valor de retorno* al proceso padre de éste.

Si un proceso ha creado mediante la llamada `fork` uno o varios hijos, la llamada `wait` suspende el proceso padre hasta que alguno de sus hijos termine su ejecución y devuelve dicho valor.

Si un proceso padre muere o termina antes que alguno de sus hijos éstos son *adoptados* por el proceso `init`, cuyo PID es 1. Es decir, a partir de ese momento el PPID de todos esos procesos es 1.

Durante el tiempo entre que un proceso muere y que su padre recoge ese valor de retorno el proceso hijo se considera un proceso *zombi*. El proceso zombi, aunque ya no consume tiempo de CPU sigue existiendo en la tabla de procesos de la máquina y no desaparecerá hasta que su proceso padre recoja su código de retorno.

WAIT (2)

SINOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>
```

```
pid_t wait(int *status);
```

DESCRIPCIÓN

La llamada wait espera hasta que muera un proceso hijo. En el parámetro statusp se devuelve el status de salida del proceso hijo.

RETORNO

Retorna el pid del proceso hijo que ha terminado o -1 si hay algún error.

5.5.- Terminación de un proceso (`exit`)

La llamada `exit` finaliza la ejecución de un proceso indicando el status de finalización del mismo.

```
EXIT (2)

SINOPSIS
     void exit (int status)

     int status;

DESCRIPCIÓN
     La llamada exit finaliza la ejecución de un proceso.
```

6.- Interrupciones software: señales

6.1.- Conceptos generales

Una señal es una interrupción software, un evento que debe ser procesado y que puede interrumpir el flujo normal de un programa.

Las señales en LINUX son el mecanismo que ofrece el kernel para comunicar eventos de forma asíncrona. Pero el kernel no es el único que puede enviar señales; cualquier proceso puede enviar a otro proceso una señal, siempre que tenga permiso.

Una alarma es una señal que es activada por los temporizadores del sistema.

Cuando un proceso se prepara para la recepción de una señal, puede realizar las siguientes acciones:

- Ignorar la señal
- Realizar la acción asociada por defecto a la señal
- Ejecutar una rutina del usuario asociada a dicha señal.

6.2.- Algunas señales

Nombre	Comentarios
SIGHUP	Colgar. Generada al desconectar el terminal.
SIGINT	Interrupción. Generada por teclado.
SIGILL	Instrucción ilegal. No se puede capturar.
SIGFPE	Excepción aritmética, de coma flotante o división por cero.
SIGKILL	Matar proceso. No puede capturarse, ni ignorarse.
SIGBUS	Error en el bus.
SIGSEGV	Violación de segmentación.
SIGPIPE	Escritura en una pipe para la que no hay lectores.
SIGALRM	Alarma de reloj.
SIGTERM	Terminación del programa.

6.3.- Capturar señales (subrutina `signal`)

La llamada *`signal`* asocia una acción determinada con una señal.
Su descripción es la siguiente:

```
SIGNAL (2)

SINOPSIS
      #include <signal.h>

      void (*signal(int signum, void (*handler)(int)))(int);

DESCRIPCIÓN
      Especifica la respuesta de un proceso ante una señal.

      signum: es el número de señal para la cual se
           especifica la respuesta.
      handler: es el tipo de respuesta deseado:
           SIG_DFL: respuesta por defecto.
           SIG_IGN: ignorar.
           nombre (puntero) de una función que es el handler
            (función que se ejecuta al recibir dicha señal).

RETORNO
      Retorna el puntero de la función que había antes si no
         hay error y SIG_ERR si hay algún error.
```

Nota: Hay que destacar que la señal puede interrumpir la ejecución de una función o llamada al sistema. Al finalizar la ejecución del handler puede que continúe la ejecución de la función, se reinicie o termine con un error (depende de la función y de la implementación concreta de LINUX). Otro aspecto a tener en cuenta es que las funciones o llamadas que se empleen en el handler han de ser reentrantes, es decir, no es seguro emplear cualquier llamada en el hanlder (por ejemplo, las funciones `malloc` y `free` no son reentrantes, si se emplean en el handler y éste ha interrumpido otro `malloc` se puede corromper la memoria del programa).

7.- Comunicación entre procesos

Normalmente, las aplicaciones que se desarrollan sobre el sistema operativo LINUX constan de varios procesos ejecutándose de forma concurrente. Por lo tanto surgen necesidades a la hora de la programación:

- Compartir información entre los procesos.
- Intercambio de información entre los procesos.
- Sincronización entre los procesos.

7.1.- Conceptos generales

Un descriptor de fichero es un número entero positivo usado por un proceso para identificar un fichero abierto.

Se llama redireccionar a establecer copias del descriptor de fichero de un archivo para encauzar las operaciones de E/S hacia otro fichero.

7.1.1.- Redirección (subrutinas dup y dup2)

La función de estas llamadas al sistema es la de duplicar un descriptor de fichero. La descripción es la siguiente:

```
DUP(2)


SINOPSIS
       #include <unistd.h>

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


DESCRIPCIÓN
       Duplica un descriptor de fichero.
       dup duplica el descriptor oldfd sobre la primera entrada de
```

la tabla de descriptores del proceso que esté vacía.
dup2 duplica del descriptor oldfd sobre el descriptor newfd. En el cado en que éste ya hiciera referencia a un fichero, lo cierra antes de duplicar.

RETORNO

Ambas rutinas devuelven el valor del nuevo descriptor de fichero y −1 en caso de error.

7.1.1.1.- Ejemplo de programa donde se utilice la llamada dup2

```
/* dup2.c - Redirección usando dup2 */
/* Ejecuta el comando que se incluya como segundo argumento
redireccionando su salida estándar hacia el fichero de nombre el
primer argumento */
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main (int contargs, char *args[])
    {
    int desc_fich;

    if (contargs < 3)
       {
       printf ("Formato: %s fichero comando [opciones].\n",
           args[0]);
       exit (1);
       }
    printf ("Ejemplo de redirección.\n");
    desc_fich = open (args[1], O_CREAT|O_TRUNC|O_WRONLY, 0666);
    if (desc_fich==-1) exit(-1);
    dup2 (desc_fich, STDOUT_FILENO); /* Redirige la salida estándar */
    close (desc_fich); /* Cierra el descriptor, ya no es necesario */
    execvp (args[2], &args[2]);        /* Ejecuta comando */
    exit (1);
    }
```

7.2.- Pipes

Las técnicas más comunes de comunicación entre procesos son los ficheros, las pipes y las named pipes (FIFOs), presentes desde las primeras versiones de LINUX.

Los ficheros permiten compartir gran cantidad de información a los procesos. La desventaja es la falta de eficiencia debido a la necesidad de acceder al disco.

Con el nombre de pipe o fifo se conoce a un mecanismo de comunicación entre procesos que provee LINUX con las siguientes características: es un "canal" de entrada/salida de datos en el que se puede escribir y leer, permite que 2 o más procesos envíen información a otro.

Vamos a explicar en detalle las pipes.

Las pipes o unnamed fifos sólo pueden ser empleadas entre procesos relacionados (pare-hijo, hijo-hijo). Es el tipo de comunicación que se emplea cuado en una shell se encadenan comando con el carácter "|".

La creación de una unnamed fifo se realiza mediante la llamada `pipe`. Su descripción es la siguiente:

PIPE (2)

SINOPSIS

```
#include <unistd.h>

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

DESCRIPCIÓN

Crea un canal de comunicación. fildes al retorno contiene dos file descriptors, fildes[0] contiene el descriptor de lectura y fildes[1] el de escritura.
La operación de lectura en fildes[0] accede a los datos escritos en fildes[1] como en una cola FIFO (primero en llegar, primero en servirse).

RETORNO

Retorna 0 si no hay error y -1 si hay algún error.

Una vez se dispone de la pipe se puede emplear como descriptores normales, es decir, se puede escribir con `write`, leer con `read` y cerrar cualquiera de ellos con `close`. Al leer del descriptor de la pipe para lectura se obtendrá `EOF` (fin de fichero) cuando se hayan cerrado TODOS los descriptores de escritura que hagan referencia a esa pipe, sean del proceso que sean.

7.2.1.- Ejemplo de utilización de la llamada `pipe`

```
/* pipe.c - pipe entre procesos padre e hijo */
#include <stdlib.h>
#include <unistd.h>

#define LEER            0
#define ESCRIBIR        1

int main ()
    {
    int descr[2];      /* Descriptores de E y S de la pipe */
    int  bytesleidos;
    char mensaje[100], *frase="Veremos si la transferecia es
         buena.";
    printf ("Ejemplo de pipe entre padre e hijo.\n");

    pipe (descr); /* Crea la pipe */

    if (fork () == 0) /* Crea un nuevo proceso. Este nuevo
       proceso tiene una copia de las variables del padre y por
       tanto tiene acceso a la pipe mediante dos entradas en SU
       tabla de descriptores de fichero */
       { /* Código ejecutado por el hijo. Es el que va a escribir. */
      /* El hijo no va a leer así que cierra su descriptor de lectura */
       close (descr[LEER]);
       write (descr[ESCRIBIR], frase, strlen(frase)+1);
      /* Cierra el descriptor de escritura que ya no necesita para que
            el otro proceso sepa que no va a haber más escrituras */
       close (descr[ESCRIBIR]);
       }
    else
       { /* Código que va a ejecutar el proceso padre */
     /* Cierra el descriptor de escritura dado que no lo va a emplear.
          Si no lo cerrase quedaría un descriptor para escribir en la
          pipe abierto, por lo que no se leería EOF nunca */
       close (descr[ESCRIBIR]);
       bytesleidos = read (descr[LEER], mensaje, 100);
```

```
        printf ("Bytes leidos: %d\n", bytesleidos);
        printf ("Mensaje: %s\n", mensaje);
        close (descr[LEER]);
        }
    }
```

7.3.- Named pipes (FIFOs)

Las named pipes permiten comunicar procesos no relacionados, existen en el sistema de ficheros, han de ser creadas utilizando `mknod` y existen hasta que se borren con `rm`.

Al igual que en las pipes, la lectura y escritura se realiza con las llamadas `read` y `write`. La apertura se realiza mediante la llamada `open`.

Como ventaja sirven para que se puedan comunicar procesos que no estén relacionados.

7.4.- System V IPCs

Se conocen como System V IPCs a tres técnicas de comunicación entre procesos que provee el LINUX System V:

- Memoria compartida: provee comunicación entre procesos permitiendo que éstos compartan zonas de memoria.
- Semáforos: dota a los procesos de un mecanismo de sincronización, generalmente para coordinar el acceso a recursos. Cuando un proceso intenta ocupar un semáforo, éste puede encontrarse en uno de los siguientes estados:
  - libre: entonces el kernel permite que ocupe dicho semáforo y el proceso continúa su ejecución.
  - ocupado: entonces generalmente el kernel pone el proceso en estado durmiente hasta que se libere el semáforo.
- Colas de mensajes: permiten tanto compartir información como sincronizar procesos. Un proceso envía un mensaje y otro lo recibe. El kernel se encarga de sincronizar la transmisión/recepción.

8.- Otras llamadas

8.1.- Obtener la hora del sistema

• **time:** retorna el valor en segundos del tiempo transcurrido desde el 1 de Enero de 1970 a las 00:00:00 GMT

• **gettimeofday:** obtiene la hora del sistema teniendo en cuenta la hora local. Trabaja con resolución de microsegundos.

8.2.- Tratamiento de errores

Existe una función de librería que permite mostrar información sobre el último error que se ha producido en un proceso como resultado de una llamada al sistema.

Su descripción es la siguiente:

PERROR (3)

SINOPSIS

```
#include <stdio.h>

void perror(const char *s);
```

DESCRIPCIÓN

La función perror produce un mensaje en la salida de error standard, describiendo el último error producido como resultado de una llamada al sistema o una llamada a una función de librería. Se imprime la cadena s y a continuación el mensaje sobre el error.

8.3.- Otras

- **chdir** Permite cambiar el directorio de trabajo del proceso
- **getenv** Permite acceder al contenido de las variables de entorno del proceso.
- **setenv** Permite modificar las variables de entorno del proceso.

Bibliografía

- G. Glass, *LINUX For Programmers And Users A Complete Guide*, Ed. Prentice Hall, ISBN 0-13-061771-7.
- W. Richard Stevens, *Advanced Programming In The LINUX Environment*, Ed. Addison-Wesley, ISBN 0-201-56617-7.
- N.Matthew & R.Stones, *Beggining Linux Programming*, Ed.Wrox, ISBN 1-874416-68-0
- C. Brown *, LINUX Distributed Programming*, Ed. Prentice Hall

PRÁCTICAS DE

SISTEMAS OPERATIVOS

Práctica 0 : INTRODUCCIÓN AL INTÉRPRETE DE COMANDOS (SHELL)

1.- Introducción

El objetivo de esta práctica es introducir al alumno en el manejo de las herramientas que va a emplear a lo largo del curso para realizar las sesiones prácticas correspondientes a la asignatura *Sistemas Operativos*. Con tal motivo, se introduce el uso del sistema operativo Linux y del lenguaje de programación C.

2.- Conceptos básicos de Linux

En este apartado se listan algunos de los comandos que pueden resultar de interés a la hora de realizar las prácticas.

Utilidades de compresión: tar, gzip, gunzip, unzip, zip.

tar : Permite adjuntar y comprimir archivos. Se recomienda el uso de *tar czfv <destino> <origen>*.

gzip, zip : Permiten comprimir archivos. Se recomienda el uso de *gzip <destino> <origen>*.

gunzip, unzip : Permiten descomprimir archivos en formato ZIP. Se recomienda el uso de *gunzip <destino> <origen>* o *unzip <destino> <origen>*.

Manejo de procesos: ps, top, kill, killall.

ps : Lista los procesos que se están ejecutando actualmente en el sistema. Se recomienda el uso de la construcción *ps axu |more* para una mejor visualización de la información, aunque también existen otras alternativas de interés.

top : Lista los procesos con un mayor consumo de recursos que se están ejecutando actualmente en el sistema. Si se desea emplear una herramienta gráfica, se recomienda el uso de *gtop* o de *ktop*.

kill : Permite matar un proceso. Este comando es útil cuando un proceso ha quedado zombie o presenta un comportamiento no deseado.

Se recomienda el uso de *kill –9 <num_proceso>* en caso de querer eliminar de forma incondicional el proceso, o de *kill –HUP <num_proceso>* en caso de querer reiniciar dicho proceso.

killall : Es similar al anterior, pero se aplica a un conjunto de procesos.

Manejo de ficheros: ls, cd, cp, mkdir, mv, pwd, rm, rmdir, chmod, chown, touch, ln.

ls : Lista el contenido de directorios del sistema. Es lo que el usuario de MS-DOS conoce como DIR. Hay muchas opciones para este comando (*-a, -l, -d, -r,...*), que a su vez se pueden combinar de muchas formas. Dado que esto puede variar según el sistema Unix en el que se esté, es recomendado consultar su página de manual (*man ls*). Sin embargo, de todas éstas, las que podríamos considerar más comunes son:

-l (*long*): Formato de salida largo, con más información que utilizando un listado normal.

-a (*all*): Se muestran también archivos y directorios ocultos.

-R (*recursive*): Lista recursivamente los subdirectorios.

Por ejemplo:

```
MathLand:~/Practicas_MNEDP/Practica1$ ls -l
total 6
drwxr-xr-x  2 lorna    users      1024 Dic 15 04:30 TeX
drwxr-xr-x  2 lorna    users      1024 Dic 15 04:30 Datos_GNUPlot
drwxr-xr-x  2 lorna    users      1024 Dic 28 04:30 Datos_Programa
-rwxr-x---  1 lorna    users    230496 Nov 12 03:08 ascii
-rw-r-----  1 lorna    users      6246 Nov 12 03:07 ascii.cpp
-rw-r-----  1 lorna    users     12920 Nov 19 03:28 graficos.cpp
MathLand:~/Practicas_MNEDP/Practica1$
```

La primera columna indica las ternas de permisos de cada fichero o directorio, más un primer carácter especial que indica, entre otras cosas, si el archivo listado es un directorio, como vemos en el ejemplo, que tenemos una d en los directorios y simplemente - si es un archivo normal. El segundo campo hace referencia al número de enlaces del archivo, mientras que los dos siguientes indican el propietario y el grupo al que pertenece. El quinto campo corresponde al tamaño del fichero en bytes, y los siguientes dan la fecha y hora de la última modificación del archivo. Por fin, la última columna indica el nombre del fichero o directorio.

cd : Con este comando podremos cambiar de directorio de trabajo. La sintaxis básica es la siguiente:

cd [nombre_directorio]

Si usamos el *shell* bash, (que es el que se instala por defecto en los sistemas Linux), simplemente tecleando cd, volvemos a nuestro directorio HOME. Si le pasamos como parámetro una ruta (que puede ser absoluta, es decir, el nombre completo desde el directorio raíz, o relativa a dónde estamos) de un directorio, cambiaremos a ese directorio, siempre y cuando tengamos permiso para entrar. Es muy posible que en nuestro Linux, al hacer cd para entrar en algún directorio, no lo veamos reflejado en el *prompt* (como sucede en MS-DOS). Para poder ver en qué directorio estamos, podemos usar el comando pwd (**p**rint **w**orking **d**irectory).

Algunos ejemplos de cd:

```
MathLand:~# cd Programas/C
MathLand:Programas/C# cd
MathLand:~#
MathLand:~# cd /usr/bin
MathLand:/usr/bin#
```

cp : Con el comando cp copiamos un archivo (origen), en otro lugar del disco (puede ser un archivo o un directorio), indicado en destino.

Su sintaxis es *cp <origen><destino>*. Si el destino es un directorio, los archivos de origen serán copiados dentro de él. Hay que decir que los *shells* de Unix aceptan *wildcards* (comodines, los trataremos en otro capítulo), que son los caracteres especiales * y ?. Así, una orden como

```
MathLand:~# cp *.html Directorio_HTML
```

copiará todos los archivos que finalicen en .html en el directorio Directorio_HTML, si éste existe. Debemos recordar que en Unix el campo "." **no** separa el nombre de la extensión de un fichero, como en MS-DOS, sino que es simplemente un caracter más. No debemos olvidarlo, sobre todo con comandos como rm, si no queremos llevarnos sorpresas desagradables. Para copiar de forma recursiva (es decir, también subdirectorios) podemos usar la opción *-r* .

mkdir : Tal y como su nombre parece querer decir, crea un directorio. La sintaxis será simplemente *mkdir <nombre_directorio>*. Para crear un directorio tenemos que tener en cuenta los permisos del directorio en que nos encontremos trabajando pues, si no tenemos permiso de escritura, no podremos crear el directorio.

mv : Con este comando podemos renombrar un archivo o directorio, o mover un archivo de un directorio a otro. Dependiendo del uso que hagamos, su sintaxis variará. Por ejemplo:

mv <archivo/s> <directorio> moverá los archivos especificados a un directorio, mientras que con *mv <archivo1><archivo2>* renombrará el primer fichero, asignándole el nombre indicado en *<archivo2>*.

Veamos unos ejemplos:

MathLand:~# mv Hola_Mundo.c Copia_Hello.c
(Renombra el archivo Hola_Mundo.c como Copia_Hello.c)
MathLand:~# mv *.c Un_Directorio
(Mueve todos los archivos finalizados en .c al directorio Un_Directorio)

pwd : Imprime en pantalla la ruta completa del directorio de trabajo donde nos encontramos actualmente. No tiene opciones, y es un comando útil para saber en todo momento en qué punto del sistema de archivos de Unix nos encontramos.

rm : Elimina archivos o directorios. Sus tres opciones son *-r* (borrado recursivo, es decir, de subdirectorios), *-f* (no hace preguntas acerca de los modos de los archivos), y *-i*(interactivo, solicita confirmación antes de borrar cada archivo). Su sintaxis es muy sencilla: *rm [-r] [-f] [-i] <archivo>*. Hay que tener mucho cuidado con este comando cuando se usen comodines, sobre todo si no lo ejecutamos en modo interactivo.

rmdir : Borra directorios *únicamente* si están vacios. Su sintaxis básica será *rmdir <directorio>*. Si queremos borrar directorios que no estén vacíos, hemos de utilizar el comando *rm -r <directorio>*.

chmod : Con este comando, cambiamos los permisos de acceso del archivo o del directorio que le especifiquemos como argumento. Podemos ejecutar de dos formas básicas este comando: la primera es *chmod <modo> <archivo>*, siendo *modo* un valor numérico de tres cifras octales que describe los permisos para el archivo. Cada una de estas cifras codifica los permisos para el dueño del archivo, para los usuarios que pertenecen al mismo grupo que el dueño, y para el resto de los usuarios, en este orden. Veamos un ejemplo: supongamos que queremos cambiar los permisos de un archivo llamado Algo.txt (por poner algún nombre), de manera que nosotros podamos leerlo y modificarlo, pero no ejecutarlo, los usuarios de nuestro grupo puedan leerlo y el resto de usuarios no pueda ni leer el fichero. La terna de permisos que describe esta situación es:

rw- r-- ---

Si escribimos un 1 en el lugar del permiso activado y un 0 en el del permiso desactivado, tenemos:

rw- r-- ---
110 100 000

Y si ahora codificamos los tripletes en binario a octal, tenemos que el permiso será 640, por tanto, tendremos que escribir el comando

```
chmod 640 Algo.txt
```

La segunda forma es algo más complicada:

```
chmod <who> +|- <permiso> <archivo>
```

Indicaremos, en el parámetro *who*, la identidad del usuario/s cuyos permisos queremos modificar *(u-user, g-group, o-others)*; a continuación irá un + o un -, dependiendo de si reseteamos el bit correspondiente o lo activamos, y en permiso debemos colocar el permiso a modificar *(r-read, w-write, x-exec)*.

Veamos unos ejemplos de ambos tipos de cambio de permisos:

```
MathLand:~# chmod 123 Ejercicio2.txt
MathLand:~# chmod 765 Practica.tex
MathLand:~# chmod g+r Un_programa.o
MathLand:~# chmod o+rx Otro_programa.p
```

chown : Con este comando, cambiamos los propietarios del archivo o directorio que le especifiquemos como argumento. Se puede cambiar no sólo el propietario del archivo, sino también el grupo al que pertenece.

chown <usuario>.<grupo> <archivo>

Un ejemplo:

```
MathLand:~# chown taxista.transportista coche.sp
```

touch : Actualiza la fecha de modificación de un archivo, o crea un archivo vacío si el fichero pasado como parámetro no existe. Con la opción *-c* no crea este archivo vacío. Su sintaxis es *touch* [*-c*] <*archivo*>.

ln : Enlaza dos archivos. Mediante *ln –s <destino> <origen>* se establece un enlace simbólico entre el archivo de destino y el de origen, de modo que el contenido al que apuntarán ambos ficheros será el mismo.

Tratamiento de ficheros: cat, file, more, less, last, tail, head.

cat : Concatena y muestra el contenido de archivos. La salida de la orden cat será por defecto la pantalla. Veamos un ejemplo:

```
MathLand:~# cat Hola_Mundo.c
#include <stdio.h>
int main(void){
  printf("Con todos ustedes, un clásico.\n\nHola, mundo!!\n");
  return 0;
  }
MathLand:~#
```

Otro uso de cat, mucho más útil, es unir varios archivos de texto en uno solo, redireccionando la salida a un fichero. Más adelante se hablará sobre redirección, pero por el momento vamos a ver un ejemplo que aclarará esta idea:

MathLand:~# cat *.c << Varios_Programas_C

Esto copiará todos los archivos terminados en .c al archivo Varios_Programas_C.

Se pueden concatenar varios ficheros, de tal manera que se añaden los contenidos de uno tras el anterior.

cat otro_fichero > mifichero

file : proporcionainformación sobre el tipo de un archivo especificado como argumento. Si lo usamos con la siguiente opción, puede sernos bastante útil: *-f <farchivo>* nos indica que en *farchivo* están los nombres de los ficheros a examinar. La sintaxis del comando es: *file* [*-f <farchivo>*] archivo. Por ejemplo:

```
MathLand:~# file Que_sera_esto
MathLand:~# Que_sera_esto: ASCII text
```

quiere decirnos que el fichero es, casi seguramente, un fichero ASCII. Este comando no es infalible, pues hay ficheros cuyo tipo no sabe reconocer, pero es útil de cara a saber, como mínimo, si es texto plano o no lo es (y si no lo es, no usaremos un editor de texto para abrirlo...).

more : Visualiza un archivo pantalla a pantalla, no de forma continua, como hace cat. Es como cat, pero con pausas, lo que nos permite leer más tranquilamente un archivo. Al final de cada pantalla nos aparecerá un mensaje indicando --More--. Si en ese momento pulsamos *RETURN*, veremos una o más líneas del archivo; si pulsamos la barra espaciadora, veremos la siguiente pantalla, si pulsamos *b* la anterior, y si pulsamos *q* saldremos de *more*. Su sintaxis es *more <archivo>*.

passwd : El comando *passwd* se utiliza para cambiar la clave de acceso al sistema. Cuando ejecutemos este comando, tendremos que escribir la clave dos veces, y en ambas ha de coincidir. Esto nos evita que se nos asigne una clave no deseada (y, lo peor, no recordada) por culpa de un error al mecanografiarla.

wc (word count) : Cuenta el número de líneas, palabras y caracteres de un fichero.

sort : Ordena las líneas de un fichero.

more - Paginar un fichero

tail : Saca las últimas líneas de un fichero (10 por defecto) .

head : Saca las primeras líneas de un fichero (10 por defecto) .

grep (get regular expression) : Saca sólo las líneas de un fichero que contienen un patrón.

find : Busca un fichero.

Comandos de red: telnet, ftp, ssh, finger, dnslookup, ping, dnsdomainname…

Por las especiales condiciones en las que se ha instalado la red del laboratorio, estos comandos serán explicados en el laboratorio.

Otros comandos: who, whoami, last, man, diff, wich…

Se trata de comandos que pueden consultarse con la orden *man* <*nombre del comando*>.

3.- Conceptos básicos de C

A la hora de programar con C, se recomienda el uso de un buen editor de textos, como puede ser el caso de *nedit, kedit, ged*it o similares. Además, es preciso el uso del compilador gcc y si procede, de un buen depurador de código o *debugger* como ddd.

La compilación de un archivo C se realiza mediante gcc –o <nombre_destino> <fichero.c>, tal y como se detalla en la documentación sobre C que se adjunta en esta memoria.

Se recomienda el uso de *Makefile* para automatizar la tarea de compilación.

4.- Ejercicios prácticos

1. Escribir un fichero de bienvenida personalizado. Es decir, cuando se ejecute debe pedir el nombre del usuario y presentar en pantalla un mensaje de bienvenida dedicado al usuario.

2. Escribir un menú con cuatro opciones para elegir las cuatro operaciones básicas: suma, resta, multiplicación y división. Utilizar la sentencia *switch* para ello. Con cada opción se debe presentar un mensaje donde se indique la operación elegida.

3. Realizar un programa que imprima en pantalla en orden decreciente tres números enteros introducidos por teclado.

4. Un centro númerico es un número que separa una lista de números enteros (comenzando desde el 1) en dos grupos de números cuyas sumas son iguales. El primer centro numérico es el 6, que separa la lista (1, 2, 3, 4, 5) y (7, 8) cuyas sumas son ambas iguales a 15. El segundo centro numérico es el 35, y separa una lista desde 1 a 49 en dos (1 a 34) y (36 a 49). La suma de los números de cada grupo es 595. Escribir un programa que halle los centros numéricos entre 1 y N, donde N es un número, menor de 7000, que se introducirá por teclado.

Utilizar para ello una función que compruebe si un número es centro numérico o no.
Una función que devuelva la suma de los números de cada grupo.

Práctica 1: OPERACIONES DE E/S, MANEJO DE FICHEROS

1.- Introducción

El lenguaje C provee varias funciones para la edición de ficheros. Estas funciones están definidas en la librería stdio.h, y por lo general empiezan con la letra f, de *file*. Además, C provee el tipo FILE, que se usará como apuntador a la información del fichero. La secuencia de tareas que se ha de seguir a la hora de acceder a un fichero es la siguiente:

- Crear un apuntador del tipo FILE * .
- Abrir el archivo utilizando la función fopen y asignándole el resultado de la llamada a nuestro apuntador.
- Hacer las diversas operaciones (lectura, escritura, etc).
- Cerrar el archivo utilizando la función fclose.

2.- Apertura y cierre de ficheros

La función ***fopen*** sirve para abrir y crear ficheros en un sistema de ficheros. El prototipo correspondiente de ***fopen*** es:

```
FILE *fopen(const char *path, const char *mode);
```

El argumento *path* hace referencia al nombre del fichero que se desea abrir, incluyendo la ruta completa dentro del sistema de ficheros.

El argumento *mode* hace referencia a los permisos con los que se dea abrir el fichero.

- "r": abrir un archivo para lectura, el fichero debe existir.
- "w": abrir un archivo para escritura, se crea si no existe o se sobreescribe si existe.
- "a": abrir un archivo para escritura al final del contenido, si no existe se crea.
- "r+": abrir un archivo para lectura y escritura, el fichero debe existir.
- "w+": crear un archivo para lectura y escritura, se crea si no existe o se sobreescribe si existe.
- "a+": abrir/crear un archivo para lectura y escritura al final del contenido.

El valor devuelto es el puntero al fichero (File *) deseado.

La función ***fclose*** sirve para cerrar ficheros en un sistema de ficheros. El prototipo correspondiente de ***fclose*** es:

```
FILE *fclose(FILE *fp);
```

Esta función sirve para poder cerrar un fichero que se ha abierto previamente. El prototipo de esta función ***fclose*** es:

```
int fclose (FILE *stream);
```

Un valor de retorno cero indica que el fichero ha sido correctamente cerrado, si ha habido algún error, el valor de retorno es la constante EOF.

A continuación se ilustra el proceso de apertura y cierre de un proceso.

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

int main(int argc, char** argv)
{
        FILE *fp;
        fp = fopen ( "nombreFichero", "r" );
        fclose ( fp );

        return 0;
}
```

3.- Fin del fichero

La función ***feof*** sirve para determinar si el cursor dentro del archivo ha alcanzado o no el final (***e**nd **o**f **f**ile, EOF*). La función ***feof*** devuelve cero (falso) si no se alcanza el final del fichero, de lo contrario devuelve un valor distinto de cero (verdadero).

El prototipo correspondiente de feof es:

```
int feof(FILE *fichero);
```

4.- Función de vuelta al origen

La función ***rewind*** permite situar el cursor de lectura/escritura al principio del archivo.

El prototipo correspondiente de rewind es:

```
void rewind(FILE *fichero);
```

5.- Lectura de un fichero

Existen varias formas de trabajar con ficheros y diferentes funciones para hacerlo. Las funciones que podríamos usar para leer un archivo son:

- `char fgetc(FILE *archivo)`
- `char *fgets(char *buffer, int tamano, FILE *archivo)`
- `size_t fread(void *puntero, size_t tamano, size_t cantidad, FILE *archivo);`
- `int fscanf(FILE *fichero, const char *formato, argumento, ...);`

5.1.- fgetc

Esta función lee un caracter del archivo señalado con el puntero **archivo*. En caso de que la lectura sea exitosa devuelve el caracter leído y en caso de que no lo sea o de encontrar el final del archivo devuelve EOF.

El prototipo correspondiente de fgetc es:

```
char fgetc(FILE *archivo);
```

Esta función se usa generalmente para recorrer archivos de texto.

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

int main()
{
        FILE *archivo;
        char caracter;

        archivo = fopen("prueba.txt","r");

        if (archivo == NULL){

                printf("\nError de apertura del archivo. \n\n");
        } else {

            printf("\nEl contenido del archivo de prueba es \n\n");

            while (feof(archivo) == 0) {
                caracter = fgetc(archivo);
                printf("%c",caracter);
            }
        }

        return 0;
}
```

5.2.- fgets

Esta función permite leer cadenas de caracteres. Leerá hasta *tamaño* caracteres o hasta que lea un retorno de línea. En este último caso, el carácter de retorno de línea también es leído.

El prototipo correspondiente de **fgets** es:

```
char *fgets(char *buffer, int tamaño, FILE *archivo);
```

El primer parámetro, *buffer*, es el puntero a la zona de memoria en la que se almacenará el contenido leído como un vector de caracteres. El segundo parámetro es *tamaño* que es el limite de caracteres a leer para la funcion fgets. Y por último, el puntero del archivo indica de qué fichero se debe leer. Esta función permite leer una línea en una sola operación.

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

int main()
{
        FILE *archivo;

        char caracteres[100];

        archivo = fopen("prueba.txt","r");

        if (archivo == NULL)exit(1);

        printf("\nEl contenido del archivo de prueba es \n\n");
        while (feof(archivo) == 0) {
                fgets(caracteres,100,archivo);
                printf("%s",caracteres);
        }
        system("pause");
        return 0;
}
```

5.3.- fread

La función ***fread*** lee *nmiemb* elementos de datos, cada uno de *tam* bytes de largo, del flujo de datos apuntado por *flujo*, almacenándolos en el sitio apuntado por *ptr*.

```
size_t fread(void *ptr, size_t tam, size_t nmiemb, FILE *flujo);
```

fread devuelve el número de elementos (no de caracteres) leídos correctamente. Si ocurre un error o se llega al final del fichero, devuelve un número negativo (o cero). ***fread*** no distingue entre fin-de-fichero y error, así que quien llame a esta función debe emplear feof(3) y ferror(3) para determinar qué ha ocurrido.

5.4.- fscanf

La función ***fscanf*** funciona igual que *scanf* en cuanto a parámetros, pero la entrada se toma de un fichero en lugar del teclado.

El prototipo correspondiente de ***fscanf*** es:

```
int fscanf(FILE *fichero, const char *formato, argumento, ...);
```

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

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

        char buffer[100];

        fp = fopen ( "fichero.txt", "r" );

        fscanf(fp, "%s" ,buffer);
        printf("%s\n",buffer);

        fclose (fp);

        return 1;
}
```

6.- Escritura de un fichero

Existen varias funciones que permiten escribir datos en ficheros. Las funciones que podríamos usar para escribir dentro de un archivo son:

- int fputc(int caracter, FILE *archivo)
- int fputs(const char *buffer, FILE *archivo)
- size_t fwrite(void *puntero, size_t tamano, size_t cantidad, FILE *archivo);
- int fprintf(FILE *archivo, const char *formato, argumento, ...);

6.1.- fputc

Esta función escribe un carácter en el archivo indicado con el puntero ***archivo**. El valor devuelto es el carácter escrito, si la operación fue completada con éxito, y en caso contrario **EOF**.

El prototipo correspondiente de **fputc** es:

```
int fputc(int carácter, FILE *archivo);
```

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

int main ( int argc, char **argv ) {
```

```
        FILE *fp;

        char caracter;

        fp = fopen("fichero.txt", "r+");

        printf("\nIntroduce un texto al fichero: ");

        while((caracter = getchar()) != '\n')
          printf("%c\n", fputc(caracter, fp));

        fclose (fp);

        return 0;
}
```

6.2.- fputs

La función **fputs** escribe una cadena en un fichero. No se añade el carácter de retorno de línea ni el carácter nulo final. El valor de retorno es un *número no negativo* o **EOF** en caso de error. Los parámetros de entrada son la cadena a escribir y un puntero a la estructura **FILE** del fichero donde se realizará la escritura.

El prototipo correspondiente de **fputs** es:

```
int fputs(const char *buffer, FILE *archivo);
```

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

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

        char cadena[] = "Mostrando el uso de fputs en un fichero.\n";

        fp = fopen("fichero.txt", "r+");

        fputs(cadena, fp);

        fclose(fp);

        return 0;
}
```

6.3.- fwrite

Esta función está pensada para trabajar con registros de longitud constante de forma análoga a **fread**. Es capaz de escribir hacia un fichero uno o varios registros de la misma longitud almacenados a partir de una dirección de memoria determinada. El valor de retorno es el número de registros escritos, no el número de bytes. Los parámetros son: un puntero a la zona de memoria de donde se obtendrán los datos a escribir, el tamaño

de cada registro, el número de registros a escribir y un puntero a la estructura **FILE** del fichero al que se hará la escritura.

El prototipo correspondiente de **fwrite** es:

```
size_t fwrite(void *puntero, size_t tamano, size_t cantidad, FILE
                              *archivo);
```

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

void menu();
void CrearFichero(FILE *Fichero);
void InsertarDatos(FILE *Fichero);
void VerDatos(FILE *Fichero);

struct sRegistro {
   char Nombre[25];
   int Edad;
   float Sueldo;
} registro;

int main() {
        int opcion;
        int exit = 0;
        FILE *fichero;

        while (!exit) {
                menu();
                printf("\nOpcion: ");
                scanf("%d", &opcion);

                switch(opcion) {
                        case 1:
                                CrearFichero(fichero);
                        break;
                        case 2:
                                InsertarDatos(fichero);
                        break;
                        case 3:
                                VerDatos(fichero);
                        break;
                        case 4:
                                exit = 1;
                        break;
                        default:
                                printf("\nopcion no valida");
                }
        }

        return 0;
}

void menu() {
        printf("\nMenu:");
        printf("\n\t1. Crear fichero");
        printf("\n\t2. Insertar datos");
        printf("\n\t3. Ver datos");
        printf("\n\t4. Salir");
}
```

```
void CrearFichero(FILE *Fichero) {
        Fichero = fopen("fichero", "r");

        if(!Fichero) {
                Fichero = fopen("fichero", "w");
                printf("\nArchivo creado!");
        } else printf("\nEl fichero ya existe!");

        fclose(Fichero);
        return;
}

void InsertarDatos(FILE *Fichero) {
        Fichero = fopen("fichero", "a+");

        if(Fichero == NULL) {
                printf("\nFichero no existe! \nPor favor creelo");
                return;
        }

        printf("\nDigita el nombre: ");
        scanf("%s", registro.Nombre);

        printf("\nDigita la edad: ");
        scanf("%d", &registro.Edad);

        printf("\nDigita el sueldo: ");
        scanf("%f", &registro.Sueldo);

        fwrite(&registro, sizeof(struct sRegistro), 1, Fichero);

        fclose(Fichero);
        return;
}

void VerDatos(FILE *Fichero) {
        int numero = 1;

        Fichero = fopen("fichero", "r");

        if(Fichero == NULL) {
                printf("\nFichero no existe! \nPor favor creelo");
                return;
        }
        fread(&registro, sizeof(struct sRegistro), 1, Fichero);
        printf("\nNumero \tNombre \tEdad \tSueldo");

        while(!feof(Fichero)) {
                printf("\n%d \t%s \t%d \t%.2f",numero,
registro.Nombre, registro.Edad, registro.Sueldo);
                fread(&registro, sizeof(struct sRegistro), 1,
Fichero);
                numero++;
        }

        fclose(Fichero);
        return;
}
```

6.4.- fprintf

La función **fprintf** funciona igual que printf en cuanto a parámetros, pero la salida se dirige a un archivo en lugar de a la pantalla.

El prototipo correspondiente de **fprintf** es:

```
int fprintf(FILE *archivo, const char *formato, argumento, ...);
```

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


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

        char buffer[100] = "Esto es un texto dentro del fichero.";

        fp = fopen ( "fichero.txt", "r+" );

        fprintf(fp, buffer);
        fprintf(fp, "%s\n", "\nEsto es otro texto dentro del
fichero.");

        fclose (fp);

        return 0;
}
```

7.- Otra forma de hacer lo mismo

En lugar de trabajar con punteros a FILE, se puede trabajar a más bajo nivel con descriptores de ficheros. Esta forma de trabajar es compatible con tuberías, sockets… y resulta de especial interés.

7.1.- open

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

int open(const char *camino, int flags);
int open(const char *camino, int flags, mode_t modo);
```

La llamada al sistema **open()** se utiliza para convertir una ruta en un descriptor de fichero (un pequeño entero no negativo que se utiliza en las operaciones de E/S posteriores como en **read**, **write**, etc). Cuando la llamada tiene éxito, el descriptor de fichero devuelto será el descriptor de fichero más pequeño no abierto actualmente para el proceso.

El parámetro *flags* es uno de **O_RDONLY**, **O_WRONLY** u **O_RDWR** que, respectivamente, piden que la apertura del fichero sea solamente para lectura, solamente para escritura, o para lectura y escritura, combinándose mediante el operador

de bits OR (|), con cero o más macros entre las que destaca **O_CREAT (s**i el fichero no existe, será creado).

7.2.- close

Permite cerrar el fichero cuyo descriptor de fichero se pasa como argumento.

```
int close(int fildes);
```

7.3.- read

read() intenta leer hasta *nbytes* bytes del fichero cuyo descriptor de fichero es *fd* y guardarlos en la zona de memoria que empieza en *buf*.

Si *nbytes* es cero, **read()** devuelve cero y no tiene otro efecto. Si *nbytes* es mayor que SSIZE_MAX, el resultado es indefinido.

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

7.4.- write

write escribe a un descriptor de fichero. **write** escribe hasta *num* bytes en el fichero referenciado por el descriptor de fichero *fd* desde el búfer que comienza en *buf.*

ssize_t write(int *fd***, const void** **buf***, **size_t** num***);***

```
int main(){

int salida, fd;
char buffer[50];
ssize_t tamano_mensaje;

char mensaje[] = "hola\n";
tamano_mensaje = strlen(mensaje);

char ruta[]="/tmp/prueba";
fd = open(ruta, O_RDWR|O_CREAT,S_IRUSR|S_IWUSR);

salida = write (fd,mensaje,tamano_mensaje);
close (fd);

fd = open(ruta, O_RDONLY);
salida = read (fd,buffer,salida);
salida = write (1,buffer,salida);

close (fd);
return 0;
}
```

Cuestiones a resolver

1.- Construya un programa que abra un fichero cuyo nombre se pase como argumento y escriba en él "Hola mundo" tras esperar 5 segundos.
2.- Construya un programa que abra un fichero cuyo nombre se pase como argumento y lea el contenido del fichero y lo imprima por pantalla.
3.- Construya un programa que cambie las vocales en minúscula a mayúsculas.

Práctica 2: PROCESOS E HILOS

1.- Introducción

Cada proceso tiene un identificador único en Linux que se denomina *process id* o *pid*.

Al proceso que solicita la creación de un nuevo proceso se le denomina **proceso padre**, y al proceso resultante se le denomina **proceso hijo**.

Cuando termina la ejecución de un proceso, éste debe finalizar ordenada y correctamente la ejecución de sus procesos hijo, pues de lo contrario dará lugar a procesos *zombies*. Algo similar ocurre con la terminación de los hilos de un proceso.

Recuerde que los comados ps, top, kill y killall le pueden ser de utilidad en el desarrollo de esta práctica.

2.- Identificadores de proceso

Las llamadas al sistema que se emplean para obtener el identificador de proceso o pid son: **getpid()** que ofrece el pid del proceso actual; y **getppid()** que ofrece el pid del proceso padre.

Si se desea obtener el identificador de usuario o uid, se emplea la llamada al sistema **getuid().**

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

pid_t pid, ppid;         /*   Declaración de variables*/
uid_t  uid;

pid = getpid();         /* Asignación de los valores devueltos por las
                                        llamadas al sistema*/
pid = getppid();
uid = getuid();
```

Nota: pid_t y uid_t son un entero largo con el ID de proceso o usuario, respectivamente.

Ejemplo de uso:

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

int main() {
printf("PID del proceso: %ld.\n", (long)getpid());
printf("PID del proceso padre: %ld.\n", (long)getppid());
printf("UID del usuario propietario: %ld.\n", (long)getuid());

return (0);
}
```

3.- Creación de procesos

La creación de procesos se lleva a cabo con la ayuda de la llamada al sistema *fork*. El nuevo proceso creado recibe una copia exacta del espacio de direcciones del padre.

Los dos procesos (padre e hijo) continúan su ejecución en la instrucción siguiente al fork. La llamada fork crea procesos nuevos haciendo una copia de la imagen en la memoria del padre. El hijo hereda la mayor los atributos del padre, incluyendo el entorno y los privilegios. El hijo también hereda alguno delos recursos del padre, tales como los archivos y dispositivos abiertos.

El valor devuelto por la llamada la sistema fork permite diferenciar el proceso padre del hijo, ya que fork devuelve el valor 0 al hijo y el ID del proceso hijo al padre. Si se produce algún error durante la creación del nuevo proceso, la función devuelve el valor -1.

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork();
```

El siguiente código genera un árbol de procesos con el padre como nodo raíz y cinco ramas que cuelgan de él.

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

int main () {
  int i, pid;

  padre = 1;

  for (i=0; i < 5; i++)
       pid = fork();

       if (pid == -1) {
          printf("Error en la creación del proceso\n");
          exit (-1);
       }
       if (pid == 0) {                          /* Proceso hijo */

          printf("Este es el proceso hijo %d, cuyo padre es %ld\n", i,
                 (long)getppid());
          exit(999);
       } else {                                 /* Proceso padre */
          printf("Este es el proceso padre con ID %ld\n",
                 (long)getpid());
        }
     }

  return (0);
}
```

3.1. Herencia de descriptores

Cuando fork crea un proceso hijo, éste hereda la mayor parte del entorno y contexto del padre, que incluye el estado de las señales, los parámetros de la planificación de procesos y la tabla de descriptores de archivo.

Hay que tener cuidado con la herencia de los descriptores de archivos porque los procesos padre e hijo comparten el mismo desplazamiento de archivo para los archivos que fueron abiertos por el padre antes del fork.

4. Las llamadas al sistema wait, waitpid y exit

La llamada al sistema wait, no confundir con el comando wait, permite que un proceso espere a que termine la ejecución de uno de sus hijos. Esta espera puede ser por un proceso concreto (waitpid) o por uno cualquiera de sus hijos (wait). En el primer caso se pasa como argumento a la llamada waitpid el pid del proceso cuya finalización se desea esperar, mientras que si basta con la finalización de uno solo de los procesos hijos, se empleará wait.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Si el estado no es NULL, tanto wait () como waitpid () almacenan la información del estado en el puntero indicado como argumento (status). Este entero puede inspeccionarse con las siguientes macros:

1. WIFEXITED(status)
   Devuelve verdadero si el hijo terminó con normalidad (exit o final de la función main).

2. WEXITSTATUS(status)
   Devuelve el estado de salida del hijo. Son los 8 bits menos significativos devueltos por el hijo mediante la llamada al sistema exit o dentro del return final de la función main. Esta macro sólo debe emplearse si WIFEXITED devuelve cierto.

3. WIFSIGNALED(status)
   Devuelve true si el proceso hijo finalizó por una señal.

4. WTERMSIG(status)
   Devuelve el número de la señal que causó la finalización del proceso hijo. Esta macro sólo debe emplearse si WIFSIGNALED devuelve cierto.

5. WIFSTOPPED(status)
   Devuelve true si el proceso hijo fue detenido por una señal.

6. WSTOPSIG(status)
   Devuelve el identificador de la señal que provocó la parada del proceso hijo. Esta macro sólo debe emplearse si WIFSTOPPED devuelve cierto.

7. WIFCONTINUED(status)
   Devuelve true si el proceso hijo se reanudó mediante la entrega de SIGINT.

En el caso de que el hijo haya terminado cuando se solicita la espera (o simplemente no existe ningún proceso hijo) la llamada *wait* devuelve el control de inmediato.

Cuando un proceso hijo termina, la llamada wait devuelve el ID de dicho hijo al padre. En caso contrario devuelve -1.

wait(&status) es equivalente a ***waitpid(-1, &status, 0)***.

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

int main (void) {
  pid_t childpid;
  int status=0, result;

  if ((childpid = fork()) == -1) {
          perror("Error en llamada a fork\n");
          exit(1);
  } else if (childpid == 0) {
          result = getpid() < getppid() ;
          fprintf(stdout, "Soy el proceso hijo (%ld) y voy a devolver a
                  mi padre (%ld) el valor %d despues de esperar 2
                  segundos\n", (long)getpid(), (long)getppid(),result);
          sleep(2);
          exit (result);
  } else {
          while( childpid != wait(&status));
          fprintf(stdout, "Soy el proceso padre (%ld) y mi hijo (%ld)
            me ha devuelto STATUS=%d\n",(long)getpid(),
            (long)childpid,status);
  }
  return (0);
}
```

La llamada al sistema exit finaliza la ejecución de un proceso. Permite asignar el valor que se pasa como argumento a la variable de entorno correspondiente para poder averiguar cuál ha sido la causa de la terminación del proceso.

```
#include <stdlib.h>
void exit (int status);
```

Si el proceso padre del que ejecuta la llamada a exit está ejecutando una llamada a wait, se le notifica la finalización de su proceso hijo y se le envía el byte menos significativo de su *status*. Con esta información el proceso padre puede saber en qué condiciones ha terminado el proceso hijo.

Si el proceso padre no está ejecutando una llamada a wait, el proceso hijo se transforma en un proceso *zombie*.

5. La llamada exec

La llamada fork crea una copia del proceso llamante. Muchas veces es necesario que el proceso hijo ejecute un código totalmente distinto al del padre. La familia de llamadas al sistema *exec* permiten la ejecución de comandos del sistema o la ejecución de programas de usuario. Tienen la particularidad de finalizar el proceso llamante cuando concluye la ejecución del citado comando o programa.

Las llamadas exec reciben como parámetros de entrada el comando a ejecutar y todos sus argumentos. Dado que el número de argumentos puede ser variable, el último parámetro de entrada es NULL, para indicar que la cadena de argumentos ha finalizado.

Las llamadas execv (execv, execvp y execve) pasan la lista de argumentos en un array de punteros a char (**char) y son útiles cuando no se conoce el número de argumentos en tiempo de compilación. Las llamadas execl (execl, execle) pasan la lista de argumentos en los argumentos de la función, indicando el último mediante el puntero 0 (NULL).

```
#include <unistd.h>
int execl (const char *path, const char *arg0, ..., const char *argn,
  char * /*NULL*/);
int execv (const char *path, char *const argv[]);
int execle (const char *path, const char *arg0, ..., const char *argn,
  char * /*NULL*/, char *const envp[]);
int execve (const char *path, char *const argv[], char *const envp[]);
int execlp (const char *file, const char *arg0, ..., const char *argn,
  char * /*NULL*/);
int execvp (const char *file, char *const argv[]);
```

Las seis variaciones de la llamada exec se distinguen por la forma en que se le pasan los argumentos.

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

int main() {
  int status;
  printf ("Lista de procesos\n");
  if (execl ("ps", "ps", "-aux", NULL) < 0) {
          fprintf(stderr, "Error en exec %d\n", errno);
          exit(1);
```

```
  }
  printf ("Fin de la lista de procesos\n");

  exit(0);
}
```

6. Terminación de procesos

Cuando termina un proceso (correcta o incorrectamente), el sistema operativo recupera los recursos asignados al proceso terminado, actualiza las estadísticas apropiadas y notifica a los demás procesos la terminación.

Cuando un proceso termina, sus hijos huérfanos son adoptados por el proceso init, cuyo ID el 1. Si un proceso padre no espera a que sus hijos terminen su ejecución, entonces éstos se convierten en procesos *zombies* y tiene que ser el proceso init el que los libere (lo hace de manera periódica).

7. Hilos

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

main (){

  pthread_t tid;
  int misargs[2], tipohilo;
  pthread_attr_t atributos;

  void *mifuncion(void *arg);
       //Creo mi propia estructura de atributos para luego modificarla
  if (pthread_attr_init(&atributos) != 0) {
          perror("En creación de estructura de atributos.");
          exit(-1);
  }
     // Compruebo el campo contentionscope con pthread_attr_getscope()
  if (pthread_attr_getscope(&atributos, &tipohilo) != 0) {
          perror("En la obtención de atributos.");
          exit(-1);
  }

  printf("Atributo de ambito por defecto: %s\n",
        (tipohilo==PTHREAD_SCOPE_SYSTEM) ? "de núcleo" : "de usuario");
  // Cambio el ámbito en mis atributos
  if (pthread_attr_setscope(&atributos, PTHREAD_SCOPE_SYSTEM) != 0) {
          perror("En el cambio de atributos.");
          exit(-1);
  }
  if (pthread_attr_getscope(&atributos, &tipohilo) != 0) {
          perror("En la obtención de atributos.");
          exit(-1);
  }
  printf("Nuevo atributo de ambito: %s\n",
        (tipohilo==PTHREAD_SCOPE_SYSTEM) ? "de núcleo" : "de usuario");
        // Ahora ya se pueden crear los hilos con el nuevo atributo
  printf("Crear hilo...\n");
```

Cuestiones a resolver

1.- Analice y describa el funcionamiento de los cuatro programas de ejemplo (proc_01.c a proc_04.c) suministrados.

2.- Construya un programa que cree cuatro procesos, A, B, C y D, de forma que A sea padre de B, B sea padre de C, y C sea padre de D.

3.- Construya un programa que cree un árbol de procesos de tres niveles de profundidad, de modo que cada rama tenga dos procesos.

4.- Realice un programa ***ejecuta***r que lea de la entrada estándar el nombre de un programa y cree un proceso hijo para ejecutar dicho programa.

5.- Construya un programa, similar al de la cuestión 2, en el que se creen cinco hijos y cada proceso termine ordenadamente 1 segundo después de hacerlo su hijo.

Práctica 3: COMUNICACIÓN ENTRE PROCESOS: PIPES

1.- Introducción

En esta práctica se va a estudiar el uso de técnicas de comunicación entre procesos basadas en tuberías o *pipes*, así como métodos de atención a varios canales de comunicación distintos (*select*) y los efectos del empleo de operaciones de lectura y escritura bloqueantes y no bloqueantes. Las *pipes* también se denominan *tuberías sin nombre*, en contraste con las FIFO que son tuberías con nombre.

Se trata de un mecanismo de comunicación entre procesos mediante el paso de mensajes, que permite que el flujo de salida de un proceso se convierta en el flujo de entrada de un segundo proceso. Para mejorar el rendimiento, la mayoría de los sistemas operativos implementan las tuberías usando *buffers*, lo que permite al proceso proveedor generar más datos que lo que el proceso consumidor puede atender inmediatamente.

2.- *pipe*

La llamada al sistema *pipe* crea un par de descriptores de ficheros, que apuntan a una tubería, y los pone en el vector de dos elementos apuntado por *filedescriptor*, donde *filedescriptor[0]* contiene el descriptor de lectura y *filedescriptor[1]* contiene el descriptor de escritura. Aquello que se escribe en *filedescriptor[1]* es leído en *filedescriptor[0]*.

```
#include <stdio.h>
int pipe(int filedescriptor[2]);
```

Las tuberías permiten una comunicación entre procesos muy sencilla mediante el paso de mensajes, ya que el tratamiento de las tuberías es el mismo que el de los ficheros. Las primitivas *write* y *read* permiten escribir y leer, respectivamente, de una tubería. Sus principales limitaciones es que sólo pueden ser empleadas para comunicar procesos relacionados, es decir, con procesos que tengan algún ancestro en común; y que únicamente permiten la comunicación unidireccional.

Para comunicar dos procesos mediante una tubería, es necesario crear la tubería antes que los dos procesos para que éstos puedan compartir la misma tubería "heredando" sus descriptores. Por ello, el proceso creará primero una tubería y luego ejecutará la llamada *fork* para crear los nuevos procesos a intercomunicar. Como los descriptores de fichero se heredan de padres a hijos, los procesos creados tras el *fork* conocerán los descriptores de la tubería, y por tanto, podrán comunicarse entre sí.

Para evitar inconsistencias cada uno de los procesos intercomunicados por una tubería cierra el descriptor de fichero correspondiente a la operación de lectura o

escritura que no va a realizar. Después se realizan las lecturas y escrituras que se deseen y al concluir, se cierra la tubería.

```
#include <stdio.h>
int tubería[2], pid;
…
pipe(tuberia);
pid = fork();
if (pid == 0) {
          char *cadena;
          cadena = "Hola mundo";
          close(tubería[0]);
          write(tubería[1], cadena, strlen(cadena)+1);
          close(tuberia[1]);
          exit(1);
} else {
          char *cadena;
          int bytesLeidos;
      cadena = (char *) malloc(100 * strlen(char));
          close(tubería[1]);
          bytesLeidos = read(tubería[0], cadena, 100);
          printf("Mensaje leído: %s\n", cadena);
          close(tuberia[0]);
          exit(1);
}
…
```

La tubería usa un *buffer* gestionado por el sistema operativo, cuyo tamaño puede variarse. En general, las llamadas read y write son bloqueantes por defecto, de tal modo que se bloquean en lectura o en escritura hasta la finalización de la operación. Si una tubería está vacía y se intenta una operación de lectura, esta operación permanecerá bloqueda (interrumpiendo la ejecución del resto del programa) hasta que la tubería tenga algo para leer y se consume la operación de lectura. Del mismo modo, si un proceso intenta escribir en la tubería y ésta está llena, la operación de escritura se bloqueará hasta que se libere espacio en la tubería y se pueda completar la escritura. La gestión de la tubería es llevada a cabo por el sistema operativo siguiendo una política FIFO (*first input, first output*).

Si se intenta escribir cuando el extremo lector se ha cerrado se genera la señal SIGPIPE (ver sección 6 de la página 79), y cuando se cierra el extremo escritor, se recibe un EOF tras la recepción de los últimos datos.

2.1.- fcntl

Permite modificar las propiedades de los descriptores de fichero mediante el argumento *cmd*. El argumento *fd* indica sobre qué descriptor de fichero se desea actuar, *cmd* el comando que se desea efectuar, y *arg* el argumento del comando indicado.

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

F_GETFL: Lee las banderas de un descriptor de fichero.

F_SETFL: De las banderas de un descriptor, establece la parte que se corresponde con las banderas de situación de un fichero al valor especificado por *arg*. Los restantes bits (modo de acceso, banderas de creación de un fichero) de *arg* se ignoran. En Linux, esta orden sólo puede cambiar las banderas O_APPEND, O_NONBLOCK, O_ASYNC y O_DIRECT.

```
#include <unistd.h>
#include <fcntl.h>
...
int miTuberia[2];

fcntl(miTuberia[0], F_SETFL, O_NONBLOCK); // Lectura no bloqueante
fcntl(miTuberia[1], F_SETFL, O_NONBLOCK); // Escritura no bloqueante
```

3.- *fifo*: tuberías con nombre.

Las tuberías con nombre, también denominadas FIFO, permiten comunicación entre dos procesos cualesquiera, entre los que no existe relación de parentesco. Es decir, no tiene un ancestro común que pueda crear la tubería y estos procesos heredarla. Son un tipo especial de fichero, y por tanto, son persistentes.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *pathname, mode_t modo);
```

El argumento *pathname* define la ruta de la tubería con nombre que se va a crear y el argumento *modo* define la máscara de permisos del fichero.

La apertura, cierre, eliminación, lectura y escritura en una *fifo* es equivalente a las operaciones sobre ficheros (open, close, unlink, read y write).

Apertura bloqueante

- open("fifo_ejemplo", O_WRONLY)
  El proceso escritor se bloquea hasta que no haya otro proceso que abra la tubería para leer de ella.
- open("fifo_ejemplo",O_RD ONLY)
  El proceso lector se bloquea hasta que no haya otro proceso que abra la tubería para escribir en ella.

Apertura no bloqueante

En este caso se emplea en la apertura el modificador O_NONBLOCK. Si se especifica, un open de sólo lectura retorna inmediatamente. Un open de sólo escritura retorna un error si ningún proceso tiene la *fifo* abierta para lectura.

En algunas ocasiones puede que sólo exista un lector y un escritor, pero no es necesario que sea siempre así, puesto que pueden existir múltiples lectores y escritores. En este caso es necesario implementar mecanismos para coordinar el uso compartido de la *fifo* .

La constante PIPE_BUF define el número máximo de caracteres que se pueden escribir en una tubería atómicamente.

Desde la línea de comandos (s*hell*) se pueden crear tuberías con nombre con la orden ***mkfifo***.

Es muy importante eliminar la fifo cuando se termine de usarla, puesto que se trata de un elemento persistente que no se elimina del sistema al concluir la ejecución de los procesos. Para ello se puede utilizar la orden ***remove***.

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

#define NOMBREFIFO "mififo"
#define TAM_BUF 100
#define TRUE 1

int main(void)
{
  int fp;
  char buffer[TAM_BUF];
  int nbytes;

  mkfifo(NOMBREFIFO,S_IFIFO|0660,0);

  while(TRUE)
  {
    fp=open(NOMBREFIFO,O_RDONLY);
    nbytes=read(fp,buffer,TAM_BUF-1);
    buffer[nbytes]='\0';
    printf("Cadena recibida: %s \n",buffer);
    close(fp);
  }
  return 0;
}
```

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

#define NOMBREFIFO "mififo"

int main(int argc, char *argv[])
{
  int fp;
  int result=1;

  if(argc!=2) printf("uso: %s cadena \n", argv[0]);
```

```
  else if ((fp=open(NOMBREFIFO,O_WRONLY))==-1)
  perror("fopen");
  else {
    write(fp,argv[1],strlen(argv[1]));
    close(fp);
    result=0;
  }

  return result;
}
```

4.- *select()*

La función `select()` indica cuál de los descriptores de fichero especificados está listo para lectura, listo para escritura, o ha producido un error sin atender. Si esta condición es falsa para todos los descriptores de fichero especificados, *select()* se bloquea hasta que venza un *timeout* o se cumpla la condición para al menos uno de los descriptores de fichero especificados.

```
int select(int nfds, fd_set *readfds, fdset *writefds, fd_set
*errorfds, struct timeval *timeout);
```

Los parámetros de la función **select()** son los siguientes:

- **int nfds:** con el valor del descriptor de fichero más alto que queremos tratar más uno. Cada vez que se abre un fichero, la llamada *open* devuelve un descriptor de fichero que es entero. Estos descriptores suelen tener valores consecutivos a partir del 3, ya que el valor 0 suele estar reservado para la entrada estándar (*stdin*), el 1 para la salida estándar (*stdout*), el 2 para la salida estándar de error (*stderr*). y a partir del 3 se nos irán asignando cada vez que abramos algún "fichero". Aquí debemos dar el valor más alto del descriptor que queramos pasar a la función más uno.
- **fd_set *readfds:** es un puntero a los descriptores de fichero de los que se pretende averiguar si hay algún dato disponible para leer o que queremos que se nos avise cuando lo haya.
- **fd_set *writefds:** es un puntero a los descriptores de fichero de los que que se pretende averiguar si se puede escribir en ellos sin peligro. Si en el otro lado han cerrado la tubería e intentamos escribir, se nos enviará una señal SIGPIPE que hará que nuestro programa se caiga (salvo que tratemos la señal).
- **fd_set *errorfds:** es un puntero a los descriptores de fichero de los que se pretende averiguar si ha ocurrido alguna excepción.
- **struct timeval *timeout:** es el tiempo que queremos esperar como máximo. Si el valor es **NULL**, la llamada a **select()** es bloqueante y no se interrumpe hasta que suceda algo en alguno de los descriptores.

Cuando la función retorna, modifica los contenidos de los **fd_set** para indicar qué descriptores de fichero tienen algo. Por ello es importante inicializarlos completamente antes de volver a llamar a la función **select()**.

Para rellenar los *fd_set* y ver su contenido tenemos una serie de macros:

- **FD_ZERO (fd_set *):** vacía el puntero, de forma que indica que no nos interesa ningún descriptor de fichero.
- **FD_SET (int, fd_set *):** mete el descriptor que le pasamos en **int** al puntero **fd_set**. De esta forma se indica que tenemos interés en ese descriptor. Llamando primero a **FD_ZERO()** para inicializar el contenido del puntero y luego a **FD_SET()** tantas veces como descriptores tengamos, ya tenemos la variable dipuesta para llamar a **select()**.
- **FD_ISSET (int, fd_set *):** indica si ha habido algo en el descriptor **int** dentro de **fd_set**. Cuando **select()** sale, debemos ir interrogando a todos los descriptores uno por uno con esta macro.
- **FD_CLR (int, fd_set *)** elimina el descriptor dentro del **fd_set**.

```
fd_set descriptoresLectura ;

FD_ZERO (&descriptoresLectura);
FD_SET (socketServidor, &descriptoresLectura);
for (i=0; i<numeroClientes; i++)
    FD_SET (socketCliente[i], &descriptoresLectura);
...
select (maximo+1, &descriptoresLectura, NULL, NULL, NULL);
...
```

5.- Redireccionamiento de una Pipe a la E/S Estándar

Por defecto, la salida estándar (*stdout*) y la salida estándar de error (*stderr*) están direccionados hacia la pantalla del terminal, y la entrada estándar (*stdin*) corresponde al teclado. Pero en ocasiones puede ser que no nos interese que la información salga en pantalla, sino que nos interesa filtrarla o redireccionarla a un archivo para guardar la información o para un tratamiento posterior, o que la entrada a un programa sea un fichero o el resultado de la ejecución de otro. Con este fin los sistemas operativos permiten la utilización de tuberías y redirecciones.

El descriptor de fichero de valor 0 suele estar reservado para la *stdin*, el 1 para la *stdout*, el 2 para la *stderr*.

El redireccionamiento se emplea para conectar procesos a través de la Entrada/Salida estándar. Esta comunicación puede ser controlada por un proceso sin modificar el código de programa. Para hacerlo hay que cerrar el descriptor de STDIN o STDOUT y llamar a **dup** o **dup2**.

```
if ((pid = fork()) < 0) {
          error_sys("error en fork");

} else if (pid == 0)
          ...       /* código del padre */

} else { /* hijo */
     close (STDIN_FILENO);          * se cierra entrada estándar */
     if (dup2(fd[0],STDIN_FILENO) < 0)
               error_sys("error en dup2");
     close(fd[0]);

     /* ahora la entrada estándar se realiza desde la salida de la
pipe */
     ...
     execvp(...);
     exit(1);
}
```

Cuestiones a resolver

1.- Construya un programa que cree dos procesos que se comuniquen entre sí mediante tuberías e intercambien diez mensajes entre sí. Para ello, el primer proceso construirá un mensaje que contendrá un contador de secuencia del mensaje y su identificador de proceso (PID). Este mensaje se enviará al otro proceso, que leerá el mensaje, aumentará el número de secuencia e introducirá su identificador de proceso. El programa concluirá cuando se hayan intercambiado los citados diez mensajes y los dos procesos concluirán ordenadamente imprimiendo por pantalla un mensaje de despedida.

```
struct mensaje {
     int secuencia, pidEmisor;
} mensaje;
```

2.- Construya dos programas independientes que actúen como se describe en la cuestión anterior, pero que hagan uso de una tubería con nombre (FIFO) para su comunicación.

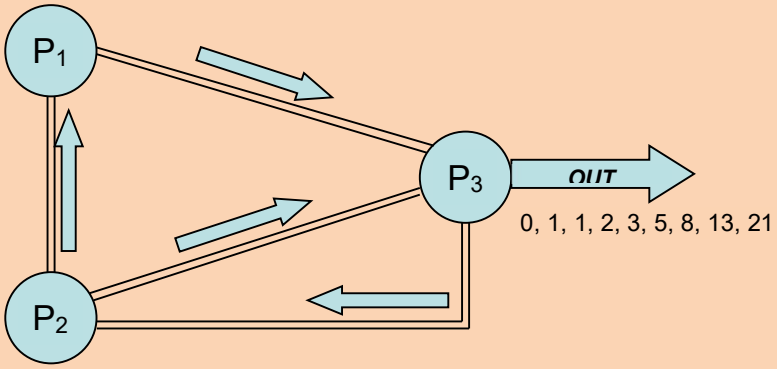
3.- Construya un programa similar al de la cuestión 1, pero empleando lecturas no bloqueantes y la función *select*.

4.- En matemáticas, la sucesión de Fibonacci es la sucesión infinita de números naturales 0,1,1,2,3,5,8,13,21…donde cada elemento es la suma de los dos **anteriores. A cada** elemento de esta sucesión se le llama número de Fibonacci.

Definición: Los números de Fibonacci f_0,f_1,f_2,f_3… quedan definidos por las ecuaciones $f_0 = 0$, $f_1=1$ y $f_n = f_{n-1} + f_{n-2}$, para n = 2, 3, 4… Es importante notar que la secuencia no tiene fin.

$$f_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f_{n-1} + f_{n-2} & \text{si } n > 1 \end{cases}$$

Escriba un programa en C que calcule la sucesión de Fibonacci empleando para ello tres procesos y sus correspondientes mecanismos de comunicación tal y como se indica en la siguiente figura. El proceso P3 irá imprimiendo en pantalla los valores obtenidos para la sucesión separados entre comas (ej.: 0, 1, 1, 2, 3, 5, 8, 13, 21) al paso de un segundo.



Práctica 4: COMUNICACIÓN ENTRE PROCESOS: SEÑALES

1.- Introducción

En esta práctica se va a estudiar el uso de técnicas de comunicación entre procesos basadas en *señales*.

Una señal (*signal*) es un a forma limitada de comunicación entre procesos empleada en Unix y otros sistemas operativos compatibles con POSIX (*Portable Operating System Interface*). Es una notificación asíncrona enviada a un proceso para notificarle un evento. Cuando se le manda una señal a un proceso, el sistema operativo modifica su ejecución normal. Si se había establecido anteriormente un procedimiento (*handler*) para tratar esa señal se ejecuta éste, si no se estableció nada previamente se ejecuta la acción por defecto para esa señal.

El usuario ya está familiarizado con su uso, ya que al escribir Ctrl-C en la shell donde se ejecuta un proceso el sistema le envía una señal SIGINT, que por defecto causa la terminación del proceso. Del mismo modo, Ctrl-Z hace que el sistema envíe una señal SIGSTOP que suspende la ejecución del proceso. Además, la llamada al sistema *kill(2)* enviará la señal especificada al proceso y el comando *kill(1)* envía la señal SIGKILL al proceso indicado como argumento. Excepciones como la división por cero o la violación de segmento también generan señales.

2.- Señales

Aunque son muchas las señales que se pueden manejar, la Tabla 1 recoge algunas de las más relevantes. Si se desea obtener más información sobre las señales disponibles y el modo de trabajar con ellas, se recomienda visitar la sección 7 del manual (*man 7 signal*).

Los manipuladores de señales, también conocidos como manejadores o *handlers*, se establecen mediante la llamada al sistema *signal(2)*. En el fondo, lo que indican es la tarea a realizar cuando se reciba la señal indicada. Si hay un manejador para una señal dada se invoca y, si no lo hay, se usa el manipulador por defecto. El proceso puede especificar también dos comportamientos por defecto sin necesidad de crear un manejador: ignorar la señal (SIG_IGN) y usar el manipulador por defecto (SIG_DFL). Las señales SIGKILL y SIGSTOP no pueden ser capturadas, bloqueadas o ignoradas.

Téngase en cuenta que las señales son asíncronas y puede ocurrir que llegue otra señal (incluso del mismo tipo) al proceso mientras transcurre la ejecución de la función que manipula la señal. Puede usarse la función *sigprocmask* para desbloquear la entrega de señales.

Las señales pueden interrumpir una llamada al sistema en proceso.

Señal	Explicación
SIGALRM	Señal de alarma, salta al expirar el timer. Reprogramable.
SIGBUS	Error en el bus *access to undefined portion of memory object* (SUS).
SIGCHLD	Proceso hijo terminado, detenido (*o que continúa). Tratamiento por defecto: ignorar. Reprogramable.
SIGCONT	Continúa si estaba parado.Tratamiento por defecto: continuar. Reprogramable.
SIGFPE	Excepción de coma flotante -- *erroneous arithmetic operation* (SUS).
SIGHUP	Hangup, al salir de la sesión se envía a los processos en Background. Tratamiento por defecto: exit. Reprogramable.
SIGILL	Instrucción ilegal.
SIGINT	Interrupción, se genera al pulsar Ctrl-C durante la ejecución. Tratamiento por defecto: exit. Reprogramable.
SIGKILL	Destrucción inmediata del proceso.Tratamiento: exit. No reprogramable, no ignorable.
SIGPIPE	Se genera al escribir sobre la pipe sin lector. Tratamiento por defecto: exit. Reprogramable.
SIGQUIT	Terminar.
SIGSEGV	*Segmentation violation*. Salta con dirección de memoria ilegal. Tratamiento por defecto: exit + volcado de memoria. Reprogramable.
SIGSTOP	Detiene el proceso. Se genera al pulsar Ctrl-Z durante la ejecución. No reprogramable, no ignorable.
SIGTERM	Terminación. Tratamiento por defecto:exit. Reprogramable.
SIGTSTP	Parada de terminal.
SIGTTIN	Proceso en segundo plano intentando leer (*in*).
SIGTTOU	Proceso en segundo plano intentando escribir (*out*).
SIGUSR1	*User defined 1*. Signal definido por el usuario.Tratamiento por defecto: exit. Reprogramable.
SIGUSR2	*User defined 2*. Signal definido por el usuario.Tratamiento por defecto: exit. Reprogramable

Tabla 1: Algunas señales relevantes.

Cada señal tiene asociado un valor que la identifica de forma unívoca, pero suele ser más inteligible su nombre. Todas las señales empiezan por el prefijo SIG y se escriben en mayúsculas puesto que son constantes que vinculan una etiqueta al valor correspondiente de la señal. La Tabla 2 ilustra esta relación y muestra también cuál es la acción por defecto que lleva a cabo el sistema cuando recibe una determinada señal.

Term	La acción por defecto es terminar el proceso.
Ign	La acción por defecto es ignorar la señal.
Core	La acción por defecto es terminar el proceso y realizar un volcado de memoria.
Stop	La acción por defecto es detener el proceso.

Señal	Valor	Acción	Explicación
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	10	Term	Term User-defined signal 1
SIGUSR2	12	Ign	User-defined signal 2
SIGCHLD	17	Cont	Child stopped or terminated
SIGCONT	18	Stop	Continue if stopped
SIGSTOP	19	Stop	Stop process
SIGTSTP	20	Stop	Stop typed at tty
SIGTTIN	21	Stop	tty input for background process
SIGTTOU	22	Stop	tty output for background process

Tabla 2: Algunas señales junto con sus valores y acciones asociadas.

3.- Envío de señales

Los procesos pueden enviar señales tanto a otros procesos como a sí mismos usando *kill(2)* por ejemplo *kill(pid,SIGUSR1)* siendo *pid* el identificador del proceso al cual deseamos enviar la señal *SIGUSR1*.

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

La llamada al sistema *kill(2)* se puede usar para enviar cualquier señal a un proceso o grupo de procesos. Si el argumento *pid* es positivo, entonces la señal *sig* se envía al proceso *pid*. La llamada devuelve 0 si se ha ejecutado con éxito, o un valor negativo en caso de fallo. Si el valor del argumento *pid* es 0, entonces el sistema envía la señal *sig* a cada uno de los procesos que forman parte del grupo de procesos del proceso actual. Si el valor de *pid* es −1, entonces se envía la señal *sig* a cada proceso, excepto al proceso 1 (*init*). Si el valor de *pid* es menor que −1, entonces se envía la señal *sig* a cada proceso en el grupo de procesos *-pid*. Si el valor de *sig* es 0, entonces no se envía ninguna señal pero se realiza la comprobación de errores.

Si la llamada concluye con éxito, la llamada devuelve el valor 0, mientras que si se produce un error, devuelve −1, y actualiza la variable *errno* apropiadamente.

Como ya se ha indicado, es imposible enviar una señal a la tarea número uno, el proceso *init*, para el que no existe un manejador de señales con el fin de que no se pueda *colgar* el sistema accidentalmente.

También se puede emplear el comando del sistema operativo *kill(1)* para enviar señales. El modo de operar es similar al indicado anteriormente, pero se lleva a cabo desde la línea de comandos (*shell*) y permite el envío simultáneo de una señal a varios procesos.

```
kill -s signal_name pid ...
kill -l [exit_status]
kill [-signal_name] pid ...
kill [-signal_number] pid ...
```

4.- Captura de señales

Todas las señales tienen una función de tratamiento por defecto. Cuando un proceso recibe una señal, suspenderá su ejecución y acudirá a la función destinada al tratamiento de esa señal y luego continuará su ejecución normal (si la señal no es de terminación). Todo esto de una forma transparente al usuario.

Es posible modificar casi todas las funciones de tratamiento de señales establecidas por defecto, pudiendo de este modo lograr que al recibir una señal el programa haga algo distinto a lo establecido por defecto.

4.1- signal

La llamada al sistema *signal()* instala un nuevo manejador de señales para la señal con número *signum*. El manejador de señales queda establecido a *sighandler* que puede ser una función especificada por el usuario o bien SIG_IGN o SIG_DFL. Cuando llega una señal con número *signum*: si el manejador correspondiente está establecido a SIG_IGN, la señal es ignorada; si el manejador está establecido a SIG_DFL, se realiza la acción por defecto asociada a la señal (ver signal(7)); si el manejador está establecido a una función *sighandler* lo primero que se hace es o bien restablecer el manejador a SIG_DFL o un bloqueo de la señal que depende de la implementación, invocando después a *sighandler* con el argumento *signum*.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

4.2- sigaction

La llamada al sistema *sigaction* se emplea para cambiar la acción tomada por un proceso cuando recibe una determinada señal. El argumento *signum* indica la señal y

puede ser cualquiera válida salvo SIGKILL o SIGSTOP. Si el argumento *act* no es nulo, la nueva acción para la señal *signum* se instala como *act*. Si *oldact* no es nulo, la acción anterior se guarda en *oldact*.

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *mask);
```

La estructura sigaction se define del siguiente modo:

```
struct sigaction {
void (*sa_handler)(int);
void (*sa_sigaction)(int, siginfo_t *, void *);
sigset_t sa_mask;
int sa_flags;
void (*sa_restorer)(void);
}
```

El elemento *sa_handler* especifica la acción que se va a asociar con *signum* y puede ser SIG_DFL para la acción predeterminada, SIG_IGN para no tener en cuenta la señal, o un puntero a una función manejadora para la señal. El elemento *sa_mask* da una máscara de señales que deberían bloquearse durante la ejecución del manejador de señal. Además, la señal que lance el manejador será bloqueada, a menos que se activen las opciones SA_NODEFER o SA_NOMASK. El elemento *sa_restorer* está obsoleto y no debería utilizarse. El elemento *sa_flags* especifica un conjunto de opciones que modifican el comportamiento del proceso de manejo de señal. Se forma por la aplicación del operador de bits OR (|) a cero o más de las siguientes constantes:

- **SA_NOCLDSTOP**

  Si *signum* es SIGCHLD, no se recibe notificación cuando los procesos hijos se paren (esto es, cuando los procesos hijos reciban una de las señales SIGSTOP, SIGTSTP, SIGTTIN o SIGTTOU).
- **SA_ONESHOT** o **SA_RESETHAND**

  Se restaura la acción para la señal al estado predeterminado una vez que el manejador de señal haya sido invocado.
- **SA_ONSTACK**

  Llama al manejador de señal en una pila de señales alternativa proporcionada por *sigaltstack(2*). Si esta pila alternativa no está disponible, se utilizará la pila por defecto.
- **SA_NOMASK** o **SA_NODEFER**

  No se impide que se reciba la señal desde su propio manejador.

Los manejadores de señal se definen una vez en el programa y la llamada al sistema signal se invoca una sola vez, no es preciso volver a invocarla cada vez que se produce

una señal. Por el contrario, las señales se pueden enviar tantas veces como se deseen con la ayuda de la llamada al sistema *kill(2).*

Cuestiones a resolver

1.- Construya un programa que cree dos procesos que se comuniquen entre sí mediante señales de modo que intercambien diez mensajes entre sí. Para ello, el primer proceso enviará la señal SIGUSR1 al segundo proceso, y éste le devolverá la señal SIGUSR2. El programa concluirá cuando se hayan intercambiado los citados diez mensajes (diez señales) y los dos procesos concluirán ordenadamente imprimiendo por pantalla un mensaje de despedida.

2.- Construya un programa que cree dos procesos. Uno de ellos simulará ser un contador, mientras que el segundo será quien lo gobierne. El proceso contador, que incrementará el valor de la variable *contador* cada segundo, se pondrá en marcha y se parará cada vez que reciba la señal SIGUSR1 del otro proceso. Además, reiniciará su contador cuando reciba la señal SIGUSR2. Cada vez que reciba la señal SIGUSR1 imprimirá por pantalla el valor del contador.
El otro proceso (gestor) enviará la señal SIGUSR1 al proceso contador cuando se introduzca un 1 por teclado y la señal SIGUSR2 al proceso contador cuando se introduzca un 2 por teclado.
La ejecución concluirá ordenadamente cuando se finalice el proceso gestor (Ctrl-C o el valor 0 por teclado) e implicará la finalización del proceso contador y después del proceso gestor.

Práctica 5: COMUNICACIÓN ENTRE PROCESOS: SEMÁFOROS Y MEMORIA COMPARTIDA

1.- Introducción

En esta práctica se va a estudiar el uso de técnicas de comunicación entre procesos mediante semáforos y memoria compartida.

La memoria compartida permite el acceso de dos o más procesos a una misma zona de memoria. Esto permite compartir datos entre dichos procesos, pero también implica garantizar el acceso en exclusiva para evitar inconsistencias en los datos mediante mecanismos como señales o semáforos. Si dos procesos tratan de acceder simultáneamente a esta área, se deberá regular su acceso para garantizar la consistencia de los datos. Este recurso IPC es el más rápido, ya que una vez conectados no necesitamos realizar más llamadas al sistema, ni interactuar con el núcleo; se trabaja directamente con el puntero que referencia a la memoria compartida.

Los semáforos son una herramienta especialmente destinada a la sincronización entre procesos. Un semáforo permite el acceso a un recurso a uno de los procesos y se lo deniega a los demás mientras el primero no concluya su tarea. Los semáforos son una interesante herramienta para gobernar el acceso a un recurso como es la memoria compartida puesto que realizan todas sus operaciones de forma atómica. Un semáforo debe garantizar la exclusión mutua (sólo va a haber un proceso en la sección crítica), que un proceso que no está en su sección crítica no pueda bloquear a otros procesos y que el proceso que está en la sección crítica no bloqueará para siempre al resto.

2.- Memoria compartida

Las utilidades de memoria compartida permiten crear segmentos de memoria a los que pueden acceder múltiples procesos, pudiendo definirse restricciones de acceso (sólo lectura, escritura). Para trabajar con un segmento de memoria compartida, es necesario crear un vínculo entre la memoria local del proceso y el segmento compartido. El proceso que vincula un segmento de memoria compartida cree estar trabajando con ella como si fuera cierta área de memoria local.

2.1.- Creación de zonas de memoria compartida

La creación de una zona de memoria compartida se lleva a cabo con la ayuda de la llamada al sistema *shmget*.

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

La llamada *shmget()* devuelve el identificador del segmento de memoria compartida asociado con el valor del argumento *key* si la operación se realiza correctamente y -1 si

se produce algún error (y *errno* recoge el número de error encontrado). El valor de error EEXIST indica que la memoria compartida ya existía y no se está creando una nueva. El argumento *key* es la clave para crear la memoria compartida (identificador de memoria) y es la misma para todos los procesos que quieran acceder a esta zona compartida de memoria. Si *key* es IPC_PRIVATE, se creará un nuevo identificador (si existen disponibilidades) que no será devuelto por posteriores invocaciones a *shmget* mientras no se libere mediante la función *shmctl*. El identificador creado podrá utilizarse por el proceso invocador y sus descendientes y por cualquier proceso que posea los permisos adecuados. Esta clave (*key*) debe ser única dentro de la máquina. Si no es así, la estructura para la comunicación entre procesos no se crea y devuelve un error. La clave puede obtenerse de tres formas:

- El servidor crea una nueva estructura especificando una clave de IPC_PRIVATE. El procedimiento creador devuelve un identificador para la nueva estructura. El problema es que ésta debe ser comunicada al proceso cliente de alguna manera.
- Otra forma es que el servidor y cliente se pongan de acuerdo en una clave. El problema es que ésta puede coincidir con otra ya existente.
- El servidor y el cliente pueden convenir un *path* y un identificador de proyecto y llamar a la función *ftok*, que convierte estos dos valores en una clave.

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

key_t ftok(const char * path, int projectID);  /* key_t es un long int */
```

Se crea entonces un nuevo segmento de memoria compartida, del tamaño indicado por el argumento *size*. El argumento *shmflg* contiene las opciones de configuración, y se efectúa un OR entre ellas. Algunas de las opciones son:

- IPC_CREAT: sirve para crear un nuevo segmento. Si no se usa este indicador, *shmget()* encontrará el segmento asociado con key y comprobará que el usuario tenga permiso para acceder al segmento.
- IPC_EXCL: sirve para asegurarse de que no existía anteriormente. Si emplea junto con IPC_CREAT, asegura el fallo si el segmento ya existe.
- Permisos: SHM_R para leer y SHM_W para escribir.

Ejemplo: se crea una zona de memoria compartida del tamaño de una variable entera.

```
int shmid;
shmid = shmget(IPC PRIVATE, sizeof(int), IPC CREAT | IPC EXCL | S IRUSR | S IWUSR);
if (shmid == -1)
perror("Error al crear la memoria compartida.");
```

2.2.- Vínculo de zonas de memoria compartida

La vinculación a una zona de memoria compartida existente se lleva a cabo con la ayuda de la llamada al sistema *shmat*.

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

```
char *shmat( int shmid, void *shmaddr, int shmflg );
```

La llamada al sistema *shmat* asocia el segmento de memoria compartida especificado por el argumento *shmid* al segmento de datos del proceso invocador. Si el segmento de memoria compartida aún no se ha asociado al proceso invocador, entonces *shmaddr* debe tener el valor 0 y el segmento se asocia a una posición en memoria seleccionada por el sistema operativo. Dicha localización será la misma en todos los procesos que acceden al objeto de memoria compartida. Si el segmento de memoria compartida ya había sido asociado por el proceso invocador, *shmaddr* podrá tener un valor distinto de cero, en ese caso deberá tomar la dirección asociada actual del segmento referenciado por *shmid*. Un segmento se asocia en modo *sólo lectura* si *shmflg* & *SHM_RDONLY* es verdadero; si no es así, se podrá acceder en modo lectura y escritura. No es posible la asociación en modo *sólo escritura*. Si la función se ejecuta con éxito, entonces devolverá la dirección de comienzo del segmento compartido, si ocurre un error devolverá -1 y la variable global *errno* tomará el código del error producido.

Ejemplo: se recupera la dirección de la zona de memoria compartida obtenida en el ejemplo anterior y coloca en ella el valor 10. Como la dirección de memoria corresponde con un valor entero, se almacena en un puntero a entero.

```
int *entero;
entero = (int *)shmat(shmid, NULL, 0);
if (entero == (int *)-1) {
     perror("Obteniendo dirección de memoria compartida");
     return -1;
}
(*entero)=10;
```

2.3.- Desvinculación de zonas de memoria compartida

La desvinculación de una zona de memoria compartida existente se lleva a cabo con la ayuda de la llamada al sistema *shmdt*.

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

char *shmdt(void *shmaddr);
```

La llamada al sistema *shmdt* desvincula el segmento de datos del proceso invocador del segmento de memoria compartida ubicado en la localización de memoria especificada por *shmaddr*. Si la función se ejecuta sin error, entonces devolverá 0, en caso contrario devolverá -1 y *errno* tomará el código del error que se haya producido.

Ejemplo: se desvincula la memoria compartida vinculada en el ejemplo anterior.

```
r = shmdt(entero);
if (r == -1) perror("Error desvinculando memoria compartida");
```

2.4.- Operaciones de control de zonas de memoria compartida

La llamada al sistema *shmctl* realiza operaciones de control en una región de memoria compartida dada.

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

int shmctl(int shmid, int cmd, struct shmid_ds *buff);
```

La llamada al sistema *shmctl* permite realizar un conjunto de operaciones de control sobre una zona de memoria compartida identificada por *shmid*. El argumento *cmd* se usa para codificar la operación solicitada (comando). Los valores admisibles para este parámetro son:

- IPC_STAT: lee la estructura de control asociada a *shmid* y la deposita en la estructura apuntada por *buff*.
- IPC_SET: actualiza los campos *shm_perm.uid*, *shm_perm.gid* y *shm_perm.mode* de la estructura de control asociada a *shmid* tomando los valores de la estructura apuntada por *buff*.
- IPC_RMID: elimina el identificador de memoria compartida especificado por *shmid* del sistema, destruyendo el segmento de memoria compartida y las estructuras de control asociadas. Si el segmento está siendo utilizado por más de un proceso, entonces la clave asociada toma el valor IPC_PRIVATE y el segmento de memoria es eliminado, el segmento desaparecerá cuando el último proceso que lo utiliza notifique su desconexión del segmento. Esta operación sólo la podrán utilizar aquellos procesos que posean privilegios de acceso a recurso apropiados para llevar a cabo esta comprobación el sistema considerará el identificador de usuario efectivo del proceso y lo comparará con los campos *shm_perm_id* y *shm_perm_cuid* de la estructura de control asociada a la región de memoria compartida.
- SHM_LOCK: bloquea la zona de memoria compartida especificada por *shmid*. Este comando sólo puede ejecutado por procesos con privilegios de acceso apropiados.
- SHM_UNLOCK: desbloquea la región de memoria compartida especificada por *shmid*. Esta operación, al igual que la anterior, sólo la podrán ejecutar aquellos procesos con privilegios de acceso apropiados.

La función *shmctl* devuelve el valor 0 si se ejecuta con éxito, o -1 si se produce un error, tomando además la variable global *errno* el valor del código del error que se haya producido. El borrado sólo es efectivo si ningún proceso tiene vinculada la memoria

compartida. Si aún existen vínculos a la memoria compartida, la operación de borrado se pospone hasta que desaparezca el último vínculo.

Ejemplo: se elimina la zona de memoria utilizada en los ejemplos anteriores.

```
r = shmctl(shmid, IPC_RMID, NULL);
if (r == -1) perror("Error desvinculando memoria compartida");
```

3.- Semáforos

Un semáforo es un mecanismo, inventado por E. Dijkstra, que permite restringir o permitir el acceso a recursos compartidos en un entorno de multiprocesamiento en el que se ejecutarán varios procesos concurrentemente.

Para utilizar los semáforos en un programa, primero se debe obtener una clave de semáforo que lo identificara unívocamente. En general, se trata de una clave de recurso compartido, ya que la función que nos sirve para obtener dicha clave también vale para memoria compartida y colas de mensajes. Para ello se utiliza, como ya se ha visto antes, la función *key_t ftok(char *, int)* a la que se suministra como primer parámetro el nombre y *path* de un fichero cualquiera que exista y como segundo un entero cualquiera. Todos los procesos que quieran compartir el semáforo, deben suministrar el mismo fichero y el mismo entero. Posteriormente, se obtiene un array de semáforos. La función ***int semget (key_t key, int nsems, int semflag)*** permite obtener dicho array de semáforos. Se le pasa como primer parámetro la clave obtenida en el paso anterior, el segundo parámetro es el número de semáforos que queremos y el tercer parámetro son *flags*.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Estos flags gobiernan los permisos de acceso a los semáforos, que son similares a los de los ficheros (lectura y escritura para el usuario, grupo y otros). También lleva unos modificadores para la obtención del semáforo. Por ejemplo, **0640 | IPC_CREATE** que indica permiso de lectura y escritura para el propietario, de lectura para el grupo y que los semáforos se creen si no lo están al llamar a *semget()*. Es importante el 0 ubicado delante del 640, ya que indica al compilador de C que cuanto sigue debe ser interpretado como un número en octal. La función *semget()* devuelve un identificador del array de semáforos.

3.1.- Inicialización del semáforo

Uno de los procesos deberá inicializar el semáforo. La función a utilizar para ello es ***int semctl(int semid, int semnum, int cmd, int ...)***. El primer parámetro (*semid*) es el identificador del array de semáforos obtenido anteriormente con *semget()*, el segundo

parámetro (*semnum*) es el índice del semáforo que queremos inicializar dentro del array de semáforos obtenido. Téngase en cuenta que el primer índice válido es el cero. El tercer parámetro (*cmd*) indica el comando que se desea aplicar al semáforo. En función de su valor, los siguientes parámetros serán una cosa u otra. El cuarto parámetro es opcional y depende de la operación que se quiera realizar (*cmd*). Si es necesario, este parámetro es una unión de tipo ***union semun***, que para el caso de usar la operación **SETVAL**, el campo *val* de dicha unión tomará el valor 1 si queremos el semáforo en "verde" o un 0 si lo queremos en "rojo".

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

En concreto, para inicializar el semáforo, el valor del tercer parámetro es **SETVAL**. Poner el valor de *semval* a *arg.val* para el *semnum*-ésimo semáforo del conjunto, actualizando también el miembro *sem_ctime* de la estructura *semid_ds* asociada al conjunto. Los procesos que se encuentran en la cola de espera son despertados si *semval* se pone a 0 o se incrementa. El proceso que realiza la llamada ha de tener privilegios de escritura en el conjunto de semáforos.

```
#include <sys/ipc.h>
#include <sys/sem.h>

union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
};

…

semun arg;
arg.val = 1; //valor inicial para el semáforo: VERDE

semctl( id_semaforo, 0, SETVAL, arg ); //configurar el semaforo 0 del array a 1
```

El comando GETVAL de *semctl*() devuelve el valor actual del semáforo.

3.2.- Empleo del semáforo

El proceso que quiera acceder a un recurso común decrementa el semáforo mediante la función ***int semop(int semid, struct sembuf *sops, unsigned nsops)***. El primer parámetro (*semid*) es el identificador del array de semáforos obtenido con *semget()*. El segundo parámetro (**sops*) es un array de operaciones sobre el semáforo. Para

decrementarlo, bastará con un array de una única posición. El tercer parámetro es el número de elementos en el citado array (*nsops*).

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, unsigned nsops);
```

La estructura del segundo parámetro contiene tres campos:

- ***unsigned short sem_num*** es el índice del array del semáforo sobre el que se deseas actuar.
- ***short sem_op*** es el valor en el que se desea decrementar el semáforo. En nuestro caso, -1.
- ***short sem_flg*** son flags que afectan a la operación. En nuestro caso, para no complicarnos la vida, pondremos 0. Algunas opciones son IPC_NOWAIT y SEM_UNDO. Si una operación ejecuta SEM_UNDO, será deshechada cuando el proceso finalice.

Al realizar esta operación, si el semáforo se vuelve negativo, el proceso se quedará "bloqueado" hasta que alguien incremente el semáforo y alcance, como mínimo, el valor 0. Cuando el proceso termine de usar el recurso común, debe incrementar el semáforo. La función a utilizar es la misma, pero poniendo un valor positivo en el campo sem_op de la estructura *strcut sembuf*. Así un valor positivo en sem_op equivale a una operación de tipo *signal* sobre el semáforo y un valor negativo equivale a una función *wait* sobre dicho semáforo. Si *sem_op* es un entero positivo, la operación añade este valor al valor del semáforo (*semval*) y si es un entero negativo, la operación resta este valor al valor del semáforo.

El conjunto de operaciones contenido en **sops* se realiza de forma atómica. El comportamiento de la llamada al sistema en caso de que no todas las operaciones puedan realizarse inmediatamente depende de la presencia de la bandera IPC_NOWAIT en los campos *sem_flg* individuales.

Si la operación se lleva a cabo correctamente, la llamada al sistema devuelve el valor 0, y en cualquier otro caso devuelve -1 con *errno* indicando el error producido.

Los semáforos binarios sólo pueden tomar dos valores (0 ó 1). Cuando el semáforo está a 1 permite el acceso del proceso a la sección crítica, mientras que con 0 bloquea el acceso a ella. Este tipo de semáforos es especialmente útil para garantizar la exclusión mutua a la hora de realizar una tarea crítica en la memoria compartida. Por ejemplo, para controlar la escritura de variables en memoria compartida, de forma que sólo se permita que un proceso esté en la sección crítica mientras que se están modificando los datos. Los semáforos n-arios pueden tomar valores desde 0 hasta N. Su funcionamiento es similar al de los semáforos binarios, ya que cuando el semáforo está a 0, está cerrado y no permite el acceso a la sección crítica. La diferencia está en que puede tomar cualquier otro valor positivo además de 1. De hecho, este tipo de semáforos son muy útiles para permitir que un determinado número de procesos trabajen

concurrentemente en alguna tarea no crítica en la memoria compartida. Por ejemplo, varios procesos pueden estar leyendo simultáneamente de la memoria compartida, mientras que ningún otro proceso intente modificar datos.

```
#include <sys/ipc.h>
#include <sys/sem.h>

void main() {
    key_t clave;
    int id_semaforo;

    clave = ftok("/bin/bash", 'X');

    id_semaforo = semget(clave, 10, 0640 | IPC_CREAT);

    sembuf accion;

    // SIGNAL sobre el semáforo
    accion.sem_num = 0; // índice del semáforo
    accion.sem_op = 1;  // operación signal (enteros positivos)
    accion.sem_flg = 0;

    semop(id_semaforo, &accion, 1);

    // WAIT sobre el semáforo
    accion.sem_num = 0; // índice del semáforo
    accion.sem_op = -1; // operación wait (enteros negativos)
    accion.sem_flg = 0;

    semop(id_semaforo, &accion, 1);
}
```

3.3.- Eliminación de semáforos

La eliminación del conjunto de semáforos y sus estructuras de datos se lleva a cabo con el concurso de la llamada al sistema *semctl()* y el comando IPC_RMID. El identificador de usuario efectivo del proceso invocador debe ser el del super-usuario, o coincidir con el del creador o propietario del conjunto de semáforos. El argumento semnum se ignora.

```
#include <sys/ipc.h>
#include <sys/sem.h>

void main() {
    key_t clave;
    int id_semaforo;

    clave = ftok("./", 'kk');
    id_semaforo = semget(clave, 10, 0640 | IPC_CREAT);

    semctl( id_semaforo, 0, IPC_RMID);   // Eliminación del grupo semafórico
}
```

Los semáforos son entidades que sobreviven a la muerte de sus procesos creadores, como ocurre con los ficheros y fifos, así que se debe prestar una especial atención a liberar los recursos una vez que éstos no vayan a ser utilizados.

Cuestiones a resolver

1.- Construya un programa que cree una zona de memoria compartida de enteros del tamaño especificado como argumento (*tamano*), que tendrá como clave identificadora la que se pase como argumento (*clave*) y que estará en ejecución hasta que se pulse Ctrl+C. Al recibir esta señal, se liberará la memoria compartida y se finalizará ordenadamente la ejecución del proceso.

```
memoria clave tamano
```

2.- Construya un programa que permita acceder a la memoria compartida identificada por el argumento clave y actualice el valor del i-ésimo entero representado por *posicion* al valor indicado por el argumento *valor*. Téngase en cuenta que el acceso puede entrar en conflicto con otras operaciones de lectura y escritura en curso.

```
escribir clave posicion valor
```

3.- Construya un programa que permita acceder a la memoria compartida identificada por el argumento clave y obtenga el valor del i-ésimo entero representado por *posicion* y lo imprima por pantalla. Téngase en cuenta que el acceso puede entrar en conflicto con otras operaciones de lectura y escritura en curso.

```
leer clave posicion valor
```

4.- Construya un programa que inicialice la memoria compartida con el valor indicado por el argumento *valor*. Téngase en cuenta que el acceso puede entrar en conflicto con otras operaciones de lectura y escritura en curso.

```
inicializar clave valor tamano
```

Lectores / escritores

Semáforos: mutex, sem_escritura;
Int num_lecturas

Write_lock
wait(sem_escritura);

Write_unlock
signal(sem_escritura);

Read_lock
wait(mutex);
num_lecturas++;
If (num_lecturas == 1) wait(sem_escritura);
signal(mutex);

Read_unlock
wait(mutex);
num_lecturas--;
If (num_lecturas == 0) signal(sem_escritura);
signal(mutex);

Práctica 6: COMUNICACIÓN ENTRE PROCESOS: COLAS DE MENSAJES

1.- Introducción

En esta práctica se va a estudiar el uso de técnicas de comunicación entre procesos mediante paso de mensajes empleando para ello colas de mensajes provistas por el sistema operativo Linux.

En la comunicación entre procesos mediante colas de mensajes, los procesos introducen mensajes en la cola y se van almacenando en ella. Cuando un proceso extrae un mensaje de la cola, extrae el primer mensaje que se introdujo y dicho mensaje se borra de la cola.

Las colas de mensajes permiten la definición de distintos tipos de mensajes, de tal forma que cada tipo de mensaje contiene una información distinta y va identificado por un entero largo, y lo que es más interesante, permite a los procesos retirar mensajes de la cola selectivamente según su tipo.

2.- Creación de colas de mensajes

A continuación se describe la forma de construir una cola de mensajes. En primer lugar se precisa una clave, de tipo **key_t**, que sea común para todos los procesos que quieran compartir la cola de mensajes. Para ello se emplea **ftok**. A dicha función se le pasa un fichero que exista y sea accesible y un entero. Si todos los procesos utilizan el mismo fichero y el mismo entero, obtendrán la misma clave.

Una vez obtenida la clave, se crea la cola de mensajes. Para ello está la función ***int msgget(key_t, int)***. Dicha función crea la cola y devuelve su identificador. Si la cola correspondiente a la *clave* ya existe, simplemente devuelve su identificador. El primer parámetro de *msgget* es la *clave* obtenida anteriormente y el segundo parámetro son unos *flags*. Aunque hay más posibilidades, lo imprescindible es:

- 9 bits menos significativos, son permisos de lectura/escritura/ejecución para propietario/grupo/otros, al igual que los ficheros. Para obtener una cola con todos los permisos para todo el mundo, debemos poner como parte de los flags el número **0777**. Es importante el cero delante, para que el número se interprete en octal y queden los bits en su sitio. El de ejecución se ignora.
- **IPC_CREAT**. Junto con los bits anteriores, este bit indica si se debe crear la cola en caso de que no exista. Si está puesto, la cola se creará si no lo está ya y se devolverá el identificador. Si no está puesto, se intentará obtener el identificador y se obtendrá un error si no está ya creada.

En resumen, los flags deberían ser algo así como 0666 | IPC_CREAT.

3.- Manejo de mensajes en las colas de mensajes

Para encolar un mensaje se utiliza la función ***msgsnd(int,struct msgbuf *,int,int)***. El primer parámetro entero es el identificador de la cola obtenido con **msgget()**.

El segundo parámetro es el mensaje en sí. El mensaje debe ser obligatoriamente una estructura cuyo primer campo sea un **long**. En dicho long se almacena el tipo de mensaje. El resto de los campos pueden ser cualquier cosa que se desee enviar (otra estructura, campos sueltos, etc). Al pasar el mensaje como parámetro, se pasa un puntero al mensaje y se le hace un *cast* a **struct msgbuf *.** No hay ningún problema en este *cast* siempre y cuando el primer campo del mensaje sea un **long**.

El tercer parámetro es el tamaño en bytes del mensaje exceptuando el **long**, es decir, el tamaño en bytes de los campos con la información. El cuarto parámetro son flags. Aunque hay varias opciones, la más habitual es poner un 0 o bien **IPC_NOWAIT**. En el primer caso la llamada a la función queda bloqueada hasta que se pueda enviar el mensaje. En el segundo caso, si el mensaje no se puede enviar, se vuelve inmediatamente con un error. El motivo habitual para que el mensaje no se pueda enviar es que la cola de mensajes esté llena.

Para desencolar un mensaje de la cola se emplea ***msgrcv(int, struct msgbuf *, int, int, int)***. El primer parámetro es el identificador de la cola obtenido con **msgget()**. El segundo parámetro es un puntero a la estructura donde se desea recoger el mensaje. Puede ser, como en la función anterior, cualquier estructura cuyo primer campo sea un long para el tipo de mensaje. El tercer parámetro entero es el tamaño de la estructura exceptuando el **long**. El cuarto parámetro entero es el tipo de mensaje que se quiere retirar. Se puede indicar un entero positivo para un tipo concreto o un 0 para cualquier tipo de mensaje. El quinto parámetro son los *flags*, que habitualmente puede ser 0 o bien **IPC_NOWAIT**. En el primer caso, la llamada a la función se queda bloqueada hasta que haya un mensaje del tipo indicado. En el segundo caso, se vuelve inmediatamente con un error si no hay mensaje de dicho tipo en la cola.

4.- Liberación de las colas de mensajes

Una vez terminada de usar la cola, se debe liberar. Para ello se utiliza la función **msgctl(int, int, struct msqid_ds *)**. Es una función genérica para el control de la cola de mensajes que permite varios comandos. En esta práctica sólo se explica cómo utilizarla para liberar la cola. El primer parámetro es el identificador de la cola de mensajes, obtenido con **msgget()**. El segundo parámetro es el comando que se desea ejecutar sobre la cola, en este caso **IPC_RMID**. El tercer parámetro son datos necesarios para el comando que se quiera ejecutar. En este caso no se necesitan datos y se pasará un **NULL**.

5.- Ejemplo de uso

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>


typedef struct mensaje_t {
   int idMensaje;
   double datoNumerico;
   char contenido[10];
} mensaje_t;

struct msgbuf {
    long mtype;
    mensaje_t mensaje;
};

int main(int argc, char **argv) {
   key_t clave;
   int idColaMensajes;
   struct msgbuf msgBuffer;

   clave = ftok ("/etc", 22);
   if (clave == (key_t) -1) exit(-1);

   idColaMensajes = msgget (clave, 0600 | IPC_CREAT);
   if (idColaMensajes == -1) exit (-2);

   msgBuffer.mtype = 2;
   msgBuffer.mensaje.idMensaje = 1;
   msgBuffer.mensaje.datoNumerico = 11.3;
   strcpy (msgBuffer.mensaje.contenido, "Mensaje");

   msgsnd (idColaMensajes, &msgBuffer, sizeof(struct msgbuf)-sizeof(long), IPC_NOWAIT);

   msgrcv (idColaMensajes, &msgBuffer, sizeof(struct msgbuf) -sizeof(long),, 2, 0);

   printf("_____\n");
   printf("ID del mensaje: %ld\n", msgBuffer mensaje.idMensaje);
   printf("Dato del mensaje: %d\n", msgBuffer mensaje.datoNumerico);
   printf("Contenido del mensaje: %s\n", msgBuffer mensaje.contenido);
   printf("_____\n");

   msgctl (idColaMensajes, IPC_RMID, 0);
}
```

Cuestiones a resolver

1.- Construya dos programas que simulen el comportamiento productor/consumidor, de tal modo que uno de ellos produzca mensajes que serán consumidos por el otro programa (hasta una total de diez). Para ello se empleará una cola de mensajes siguiendo los ejemplos anteriormente descritos y el tiempo empleado por cada proceso para producir o consumir un mensaje se pasará como argumento.

```
productor clave_cola periodo
consumidor clave_cola periodo
```

2.- Construya un programa que cree cinco instancias del programa productor del ejercicio anterior con igual periodo de producción y con distintos periodos de producción y evalúe qué es lo que ocurre en ambos casos.

Práctica 7: SHELL

1.- Introducción

El objetivo de esta práctica es construir una pequeña shell haciendo uso de una librería, de la llamada al sistema *execvp*, de las redirecciones de entrada y salida (*dup2*) y de comunicación mediante señales. Esta práctica pretende poner en contexto algunos de los conceptos introducidos en las prácticas precedentes, de modo que el alumno sea capaz de entender el funcionamiento de las herramientas de gestión y comunicación de procesos que se la han presentado anteriormente y que sea capaz de desarrollar una pequeña aplicación que emplee los conocimientos adquiridos.

El objetivo de esta práctica es construir una *shell* o intérprete de empleando las llamadas al sistema de manejo de procesos (fork, wait, exec, dup2…) que permita ejecutar cualquier comando del sistema operativo o ejecutar otro programa, que admita tuberías (|), así como redireccionamiento de entrada y salida (<, >).

2.- Especificaciones

En este apartado se especifican las funcionalidades que debe cumplir la aplicación indicada.

- **Admitir pipes** "|". Ejemplo: *more kk.txt | grep hola*, imprimirá por pantalla aquellas líneas del fichero *kk.tx*t que contengan la palabra *hola*. Para ello se crearán dos procesos, uno de los cuales ejecutará el comando *more* y el otro el comando *grep*. Ambos procesos se intercomunicarán con la ayuda de una tubería, de tal modo que la salida del primer proceso se escriba en la tubería y la entrada del segundo proceso se lea de dicha tubería.
- **Admitir redirección a fichero** "**> fichero**". Ejemplo: *ls –al > kk.txt*, listará el contenido del directorio local en el fichero *kk.txt*.
- **Admitir anexión a fichero** "**>> fichero**". Idéntico al anterior salvo por el modo de volcar a fichero. En este caso no se sobreescribe el contenido del fichero sino que se anexa al final de éste.
- **Admitir entrada desde fichero** "**< fichero**". Ejemplo: *wc -l < kk.txt*, contará el número de líneas del fichero *kk.txt*.
- **Admitir redirección de entrada y salida simultánea**.
- ***Prompt* personalizado:** *minishell\>*
- **La conclusión de la ejecución de la *Shell* se alcanzará al introducir el comando *exit* o pulsar *Ctrl+C*.**

3.- Librería

Para construir la *shell* se dispondrá de una pequeña librería con una función (*fragmenta*) que permite trocear cadenas. Esta función tomará una cadena de texto de la línea de comandos y la fragmentará de tal modo que se ajuste a la estructura de datos que posteriormente empleará para rellenar la estructura de datos que precisa la llamada al sistema *execvp* para ejecutar los comandos contenidos en la cadena de texto leída. La librería se completará con otra función (*borrarg*) que permitirá liberar la memoria reservada por la función de fragmentación.

La librería se denominará *fragmenta.o* y presentará las siguientes características:

FRAGMENTA (3) FRAGMENTA (3)

NOMBRE

fragmenta, borrarg – Utilidades para trocear cadenas.

SINOPSIS

```
#include "fragmenta.h"
char **fragmenta(const char *cadena);
void borrarg(char **arg);
```

DESCRIPCIÓN

- ***fragmenta():*** crea una array de *char** con tantos elementos como el número de fragmentos que encuentre en cadena más uno, el último vale siempre *NULL* y es el único con tal valor, con lo que sirve para determinar el final del array. Cada elemento de este array es un puntero a una zona de memoria donde se encuentra uno de los fragmentos de cadena y en el mismo orden. Los fragmentos de cadena vienen definidos por estar separados por uno o más espacios, pudiendo terminar en un fin de línea. Si la cadena a fragmentar posee múltiples espacios en blanco, ninguno de éstos se almacenará en la estructura de datos resultante.
- ***borrarg():*** libera la memoria asociada con el puntero *arg*, así como las zonas de memoria apuntadas por cada uno de los *char** apuntados por *arg* y colocados uno tras otro hasta uno que valga NULL.

VALOR DEVUELTO

fragmenta() devuelve el puntero al array creado o NULL si no puede realizar su función.

El contenido de la cabecera de la librería es:

```
/* Contenido de fragmenta.h */
char **fragmenta(const char*);
void borrarg(char **arg);
/* Fin de fragmenta.h */
```

4.- *Shell*

Para construir la *shell*, será preciso crear nuevos procesos mediante la orden *fork* y duplicar los descriptores mediante la orden *dup2*. Otras llamadas al sistema de interés son *wait, pipe, signal, open* y *close*.

La descripción de la *shell* será:

MSH (3) MSH (3)

NOMBRE

`minishell - Mini Shell`

SINOPSIS

`msh`

DESCRIPCION

Las funcionalidades de esta shell son:

- Ejecución de comandos con un número indeterminado de argumentos.
  Ejemplo:
  minishell\> cp -r sources backup
- Redirección de la salida estándar a fichero mediante>
  Ejemplo:
  minishell\> ls –al > listado
- Redirección de la entrada estándar de fichero mediante <.
  Ejemplo:
  minishell\> wc -l < listado
- Redirección simultánea de entrada y salida estándar en cualquier orden.

5.- Entrega de la práctica

En la carpeta de prácticas de MiAulario se dispone de la biblioteca fragmenta, sus fuentes y el *Makefile* correspondiente. En el fichero comprimido a entregar al concluir la práctica debe encontrarse el *Makefile* provisto, así como todos los ficheros .c y .h necesarios para crear *minishell*. La acción por defecto del *Makefile* (la cual debe funcionar con sólo hacer make en ese directorio) debe ser crear el ejecutable de la shell. Igualmente debe responder correctamente a *make fragmenta* y a *make prueba*.

Para la corrección de la práctica se borrarán todos los ejecutables, se hará un *touch* a todos los ficheros fuente y se recompilará mediante el *Makefile*.

6.- Aplicaciones, funciones y llamadas al sistema útiles

`fork(2), execvp(3), wait(2), open(2), close(2), dup2(2), pipe(2), signal(2) y kill(2).`

Práctica 8: PLANIFICACIÓN DE PROCESOS

1.- Introducción

El objetivo de esta práctica es profundizar en el conocimiento y manejo de la característica multiproceso de LINUX construyendo un planificador (*scheduler*) de procesos a alto nivel basado en señales, que implementa un sistema de colas de dos niveles APROPIATIVOS en el que los procesos de nivel 1 siguen una política de planificación FIFO, y los de nivel 2 una política de turno rotatorio (también conocida como *round robin*) con turnos de 5 segundos. Siendo los procesos de nivel 1 los de mayor prioridad, que deberán atenderse a la mayor celeridad.

2.- Planificador de procesos a nivel de usuario procsched

El planificador (*procsched*) se construirá con la ayuda de una cola de mensajes. Se dispondrá de un proceso que estará encargado de recibir las solicitudes de ejecución de procesos. Este proceso construirá las colas de mensajes que se precisen (y las eliminará ordenadamente cuando proceda), construirá la estructura de datos correspondiente para cada proceso y la encolará en la cola de mensajes que proceda, creará un segundo proceso encargado de la planificación, y liberará todos los recursos (procesos y colas de mensajes) cuando se indique su finalización por parte del usuario.

El planificador admitirá la entrada de solicitudes tanto por teclado, como por un archivo de configuración. Cuando las solicitudes se obtengan por teclado, el final del fichero (EOF) se logra pulsando Ctrl+D.

PROCSCHED(1) PROCSCHED(1)

NOMBRE

procsched - *Scheduler* de procesos a nivel de usuario

SINOPSIS

procsched configfile]

DESCRIPCIÓN

El programa *procsched* crea las colas de mensajes y el planificador.

El programa *procsched* lanza los procesos que le han sido configurados y va encolando las peticiones de ejecución de los procesos en las colas de mensajes conforme recibe las solicitudes. El planificador va ejecutando los procesos encolados en las colas de mensajes atendiendo a la prioridad de cada nivel (la máxima prioridad corresponde al nivel más bajo).

El fichero de configuración configfile es opcional, de no estar presente se leerá la información de configuración de la entrada estándar, siguiendo el mismo formato que el fichero.

Formato del fichero de configuración: Consta de una línea por cada programa a ejecutar. Cada línea es de la siguiente forma:

```
nivel nombreprograma argumento1 argumento 2 argumento 3 …
```

Donde:
- nivel: indica la cola en la que se ha de encolar cada proceso.
- nombreprograma argumento1 argumento 2 argumento 3 …: es el programa a ejecutar junto con sus argumentos, que pueden ser un número indeterminado y diferente para cada programa.

El programa *procsched* no finalizará hasta que el usuario pulse CTRL+C, momento en el que se terminará adecuadamente liberando todos los recursos empleados y se ofrecerá por pantalla el número de procesos que han concluido con éxito y el número de cambios de contexto realizados.

VALOR DEVUELTO

El planificador *procsched* indicará a su finalización el número de procesos que han concluido con éxito y el número de cambios de contexto producidos.

3.- Entrega de la práctica

En el fichero comprimido a entregar al concluir la práctica debe encontrarse el *Makefile* provisto, así como todos los ficheros .c y .h necesarios para crear *procsched*. La acción por defecto del *Makefile* (la cual debe funcionar con sólo hacer make en ese directorio) debe ser crear el ejecutable *procsched*. Para la corrección de la práctica se borrarán todos los ejecutables, se hará un *touch* a todos los ficheros fuente y se recompilará mediante el *Makefile*.

Se sugiere emplear la librería *fragmenta.o* ya empleada en la práctica anterior.

4.- Aplicaciones, funciones y llamadas al sistema útiles

kill(1), kill(2), sigaction(2), sleep(3), raise(3), signal(2), signal(7), sleep(2), nanosleep(2), pause(2), alarm(2), execvp(3), fork(2), wait(2).

Práctica 9: CONSTRUCCIÓN DE UN PEQUEÑO SISTEMA CONCURRENTE

1.- Introducción

El objetivo de esta práctica es construir de un pequeño sistema concurrente que permita poner en práctica todos los conceptos aprendidos en las prácticas anteriores. Para ello, se desea construir un sistema de control de producción mediante el empleo de procesos, mecanismos de comunicación entre procesos y llamadas al sistema.

2.- Desarrollo

Se desea construir un sistema de control de producción mediante el empleo de procesos, mecanismos de comunicación entre procesos y llamadas al sistema. Para ello se requieren cuatro programas diferentes que interactuarán entre sí conforme se indica a continuación.

```
struct mensaje {
    long idMensaje        // tipo de mensaje
    int idAnillo;         // identificador del anillo al que pertenece el mensaje
    int numVuelta;        // número de vuelta dentro del anillo
    int nodosVisitados;   // número de nodos visitados
    int ordenLlegada;     // orden de llegada del mensaje
} mensaje;
```

GESTOR

Cree el programa GESTOR que se encargará de crear una cola de mensajes, una memoria compartida y un semáforo a partir de una misma clave que se pasará como argumento (clave). El programa gestor inicializará tanto la memoria compartida (a cero) como el semáforo. Posteriormente, ejecutará el proceso informador y creará tantos anillos como se indique como argumento de entrada (numAnillos). Cada anillo tendrá tantos nodos como se indique por argumento (numNodos), y cada uno de los nodos estará enlazado mediante una tubería (pipe) con su antecesor y su sucesor. Para ello, Gestor dispondrá del correspondiente vector de tuberías que permita enlazar los nodos, hará las correspondientes redirecciones y ejecutará los procesos de tipo nodo.

Gestor permanecerá a la espera, sin ejecutar ninguna tarea, hasta que se pulse ^C, momento en el que procederá a finalizar correctamente todos los procesos (nodos, coordinadores e informador), eliminará el semáforo, eliminará la memoria compartida, eliminará la cola de mensajes y finalizará su ejecución.

gestor clave numAnillos numNodos periodo

INFORMADOR
Leerá indefinidamente, y de forma bloqueante, los mensajes que se vayan encolando por parte de los procesos coordinadores en la cola de mensajes, que habrá obtenido previamente a partir del argumento clave.
Cada vez que se desencole un mensaje, imprimirá por pantalla el siguiente mensaje:

[ordenLlegada idAnillo numVuelta nodosVisitados]

Este proceso de desencolado de mensajes e impresión por pantalla se mantendrá en el tiempo hasta que reciba de Gestor una señal de terminación, a cuya recepción procederá a finalizar su ejecución.

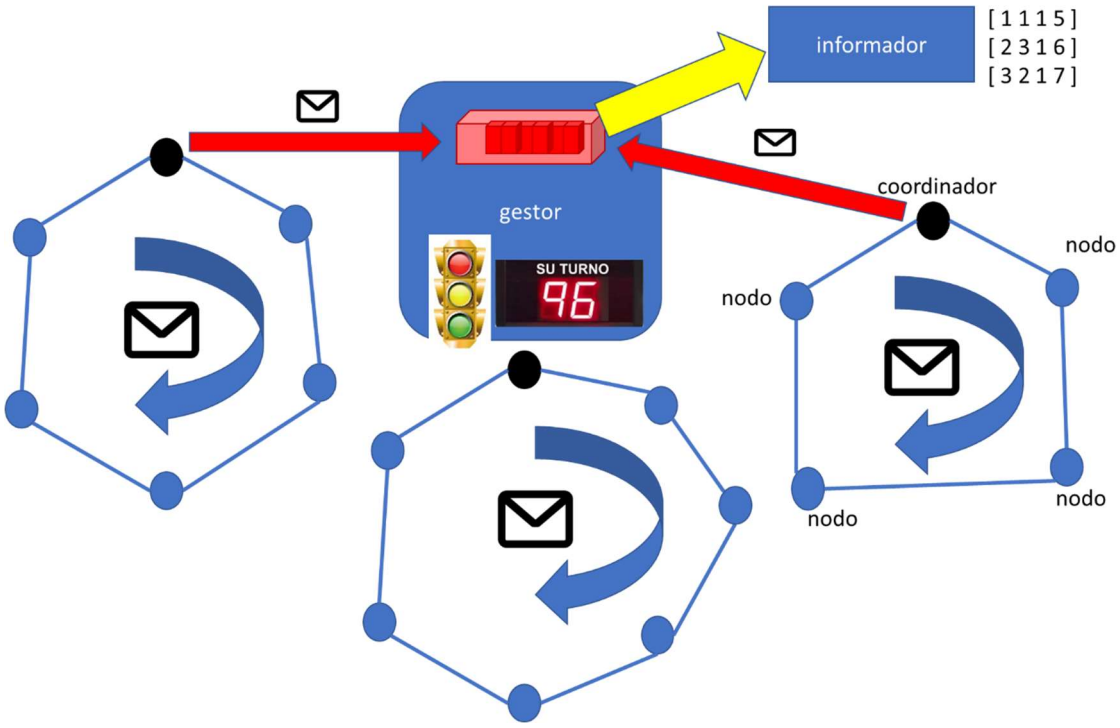
informador clave

NODO
Leerá continuamente de teclado el mensaje, esperará el tiempo esperado por el argumento periodo, e imprimirá el mensaje por pantalla. El mensaje deberá no debe transformarse ni manipularse en ningún momento.

nodo periodo

COORDINADOR
Actuará del mismo modo que los nodos (nodo) pero además será el encargado de iniciar el envío del mensaje en el anillo, de inicializar y actualizar adecuadamente el mensaje, y de encolarlo cuando proceda en la cola de mensajes.
Así, inicializará el identificador del anillo (idAnillo) que se le indicará como argumento, el número de vuelta (numVuelta, a cero) y el número de nodos visitados en dicho anillo (nodosVisitados) que también recibirá como argumento (numNodos), esperará el tiempo indicado (periodo) y enviará el mensaje a su sucesor. Cada vez que reciba el mensaje de nuevo procederá a incrementar el valor de numVuelta, bloqueará el semáforo, accederá a la memoria compartida para obtener el orden de llegada (que almacenará en ordenLlegada), incrementará el valor de la memoria compartida en una unidad), encolará el mensaje actualizado en la cola de mensajes y desbloqueará el semáforo. Este proceso se repetirá indefinidamente hasta que reciba una señal de terminación por parte de gestor. Para obtener la memoria compartida, el semáforo y la cola de mensajes empleará la clave que se le pasa como argumento.

coordinador clave idAnillo numNodos periodo



3.- Entrega de la práctica

En el fichero comprimido a entregar al concluir la práctica debe encontrarse, además de los citados cuatro programas, el *Makefile* que permita la compilación conjunta de los cinco programas, así como todos los ficheros .c y .h necesarios para su correcto funcionamiento. La acción por defecto del *Makefile* (la cual debe funcionar con sólo hacer make en ese directorio) debe ser crear los ejecutables de los cinco programas anteriormente descritos. Para la corrección de la práctica se borrarán todos los ejecutables, se hará un *touch* a todos los ficheros fuente y se recompilará mediante el fichero *Makefile*.

Se sugiere emplear todos aquellos recursos disponibles de las prácticas anteriores.

Se valorarán especialmente la concisión, claridad, simplicidad y eficiencia en el código.

Anexo I: Ejemplos de interés

Ejemplo de utilización de memoria compartida en UNIX

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

#define ARRAY_SIZE   4000
#define MALLOC_SIZE 10000
#define SHM_SIZE    10000
#define SHM_MODE    (SHM_R | SHM_W)     /* read/write */

char array[ARRAY_SIZE];  /* datos sin inicializar = bss */

int main() {
   int shmid;
   char *ptr, *shmptr;

   printf("array[] desde %x hasta %x\n", &array[0], &array[ARRAY_SIZE]);
   printf("stack sobre %x\n", &;shmid);

   if ((ptr=malloc(MALLOC_SIZE)) == NULL)
      fprintf(stderr, "error de malloc()\n");
   printf("malloc desde %x hasta %x\n", ptr, ptr+MALLOC_SIZE);

   if ((shmid=shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE))<0)
      fprintf(stderr, "error de shmget()\n");
   if ((shmptr=shmat(shmid, 0, 0)) == (void *) -1)
      fprintf(stderr, "error de shmat()\n");
   printf("shared memory desde %x hasta %x\n", shmptr, shmptr+SHM_SIZE);
   /* proceso padre */
   if (0!=fork()) {
      while (*shmptr != 'x') ;
      printf("he comprobado 'x' en memoria compartida\npulse una tecla");
      getchar(); /* comprobar el semaforo con 'ipcs' */
      /* eliminar memoria compartida */
      if (shmctl(shmid, IPC_RMID, 0) < 0)
         fprintf(stderr,"error de shmctl()\n");

      exit(0);
   /* proceso hijo */
   } else {
      *shmptr = 'x';
      exit(0);
   }
}
```

rshmem.h : Fichero de cabecera con definiciones y declaraciones para usar memoria compartida.

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

#define ARRAY_SIZE   4000
#define MALLOC_SIZE 10000
#define SHM_SIZE    10000
#define SHM_MODE    (SHM_R | SHM_W)     /* read/write */

#define TRUE      1
#define FALSE     0

#ifdef RUTINAS_SHMEM
   static int shmid; /* handler de memoria compartida */
   char *memoria;    /* puntero a zona de memoria compartida */
#else
   extern char *memoria;
#endif

/* Prototipos de funciones de memoria compartida */

void origenTiempo();
void tiempoPasa();
int crearMemoria() ;
int eliminarMemoria() ;

#define TP tiempoPasa();
```

rshmem.c : Fichero con funciones de creación de memoria compartida y varias de utilidad.

```
#define RUTINAS_SHMEM

#include "rshmem.h"

/* Crea memoria compartida.
*    - el manejador de memoria es interno
*    - manda mensajes de error por salida de error estándar.
*/
int crearMemoria() {
   char *funcName = "crearMemoria";
   if ((shmid=shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE))<0) {
      fprintf(stderr, "%s: error de shmget()\n", funcName);
   } else if ((memoria=shmat(shmid, 0, 0)) == (void *) -1) {
      fprintf(stderr, "%s: error de shmat()\n", funcName);
   } else {
      return TRUE;
   }
   return FALSE;
}

/* Destruye la memoria compartida creada por crearMemoria()
*/
int eliminarMemoria() {
   char *funcName = "eliminarMemoria";
   if (shmctl(shmid, IPC_RMID, 0) < 0) {
      fprintf(stderr,"%s: error de shmctl()\n", funcName);
      return FALSE ;
   } else
      return TRUE ;
}


/* Coloca una semilla en el temporizador del bucle de
*  tiempoPasa()
*/
void origenTiempo(){
   srand((unsigned int) time(NULL)) ;
}

/* Rutina que hace pasar un poco de tiempo con un bucle
*  sencillo
*/
void tiempoPasa() {
   unsigned int i;
   int a=3;

   /* Los parametros "50" y "2" dependen mucho de la velocidad
   *  de la computadora y de la configuracion del SO. Espero que
   *  funcionen bien
   */
   for (i=rand()/50; i>0; i--)
      a = a%3 + i;
}
```

Ejemplo de dos procesos con condición de carrera

```
#include "rshmem.h"

void incrementa(int *mem, int k) {
   int i;
   i=*mem; TP
   i=i+k;  TP
   *mem=i;
}

int main() {
   int  *recurso;
   char *marcaFin;

   /* crear zona de memoria compartida */
   if (!crearMemoria())
      fprintf(stderr, "error de crearMemoria\n");

   recurso  = (int *) memoria ;
   marcaFin =  memoria + sizeof(int) ;
   *recurso =   0 ;
   *marcaFin = 'p' ;
   if (0!=fork()) {    /* proceso padre */
      int i;
      for (i=0; i< 1000; i++)
         incrementa(recurso, -5);
      while (*marcaFin != 'x') ; /* espera al hijo */

      printf("El recurso valia 0 y ahora vale %d\n", *recurso);
      if (!eliminarMemoria())   /* eliminar memoria compartida */
         fprintf(stderr, "error de eliminarMemoria\n");
      exit(0);
   } else {            /* proceso hijo */
      int i;
      for (i=0; i< 1000; i++)
         incrementa(recurso, 5);
      /* termina */
      *marcaFin = 'x';
      exit(0);
   }
}
```

Ejemplo de destrucción de memoria compartida en UNIX

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

#define ARRAY_SIZE 4000
#define MALLOC_SIZE 10000
#define SHM_SIZE 10000
#define SHM_MODE (SHM_R | SHM_W) /* read/write */

int main() {
   int  shmid;
   char shmidStr[128];

   printf("Que zona de memoria desea destruir? ");

   shmid = atoi(gets(shmidStr));
   if (shmctl(shmid, IPC_RMID, 0))<0)
      fprintf(stderr, "error de shmctl()\n");

   exit(0);
}
```

Anexo II: Práctica de familiarización con Linux

Práctica 10: INSTALACIÓN Y CONFIGURACIÓN BÁSICA DE SISTEMA OPERATIVO: LINUX

1.- Introducción

El objetivo de esta práctica es introducir al alumno en la instalación de un sistema operativo de tipo Unix. En concreto se instalará LUbuntu 14.04 (http://lubuntu.es/). Se realizará una instalación guiada del sistema operativo sobre un sistema de virtualización (VirtualBox). Asimismo se realizará una pequeña introducción a los elementos básicos que conforman este tipo de sistemas y a tareas básicas de administración:

- Estructura del sistema de archivos
- Proceso de arranque del sistema
- Gestión de usuarios / permisos
- Gestión de software (instalación, configuración) tanto de programas como de servicios
- Administración de red
- Administración de dispositivos (impresoras, unidades de almacenamiento externo…)

2.- Instalación

Para la instalación tomaremos como referencia la guía de instalación disponible en su sitio web:

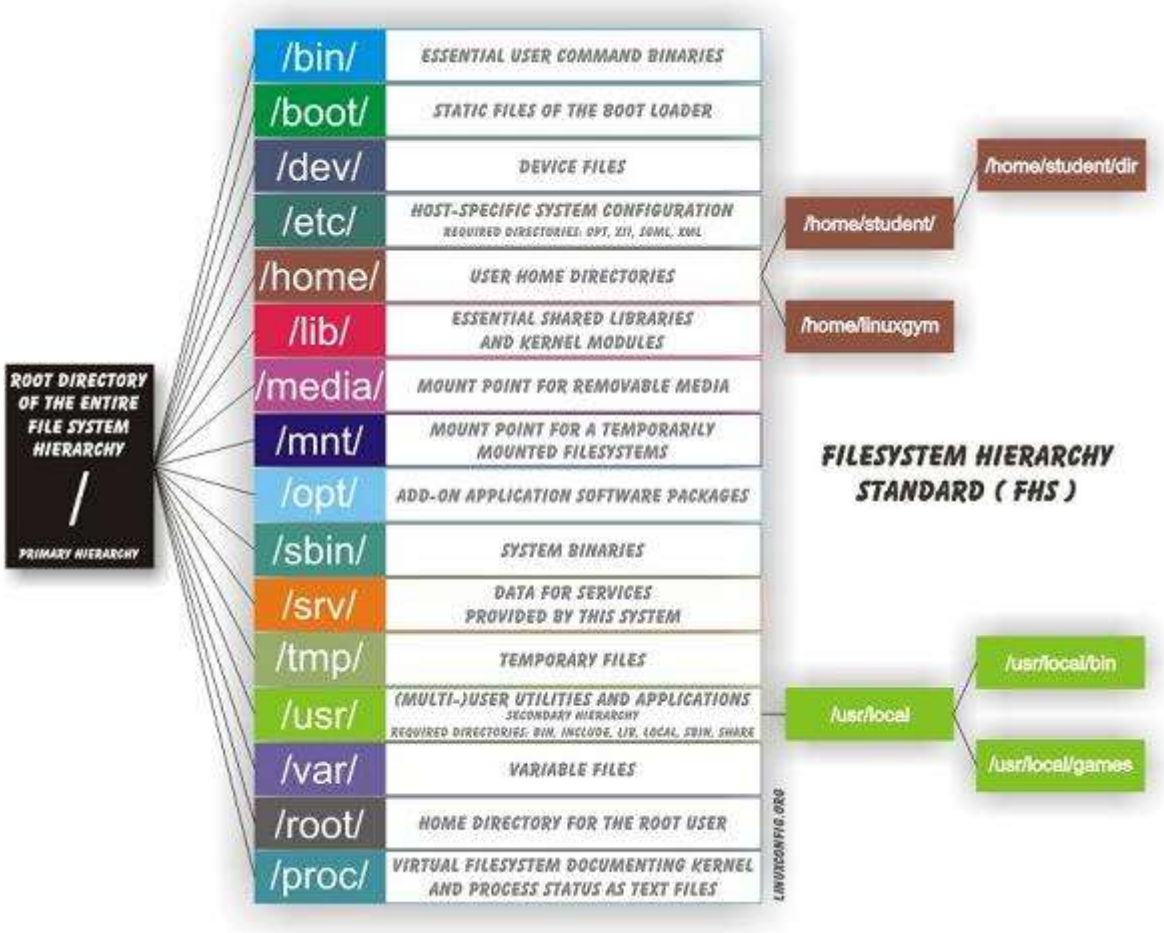
`http://lubuntu.es/instalar-lubuntu/`

Los pasos principales que seguiremos serán los siguientes:

- Revisión de los requisitos del sistema
- Medios de instalación
- Arranque de la instalación
- Particionado de discos
- Selección del software a instalar
- Arranque del nuevo sistema

3.- Estructura de ficheros

Los sitemas Unix comparten (en su gran mayoría) una estructura estándar a la hora de organizar los directorios y ficheros del sistema operativo:



En esta estructura se agrupan los ficheros siguiendo su uso/cometido.

4.- Gestión de usuarios / permisos

Los sistemas Unix implementan el concepto de usuarios/grupos y permisos de ficheros y directorios según dichos usuarios/grupos. Cada usuario del sistema accede al mismo mediante una contraseña. La información acerca de los usuarios, los grupos y las contraseñas se encuentran en los siguientes ficheros:

```
/etc/passwd      - Información de las cuentas de usuario
/etc/shadow      - Almacenamiento de las contraseñas
/etc/group       - Información de los grupos existentes en el sistema
```

Además se dispone una serie de herramientas para crear, modificar y eliminar tanto usuarios como grupos: `useradd`, `userdel`, `groupadd`, `groupdel`, `groupmod`.

Los permisos de los ficheros se estructuran en tres grupos: permisos del dueño del fichero, permisos del grupo dueño del fichero y permisos del resto de los usuarios. Los permisos básicos son lectura, escritura y ejecución. Para el caso de directorios la

ejecución permite el acceso al directorio. Un listado típico de ficheros permite conocer estos permisos:

```
usuario@host:~# ls -la /home/usuario
total 24
drwxr-xr-x    2 usuario   usuarios     4096 Mar 12 05:58 .
drwxrwsr-x    4 root      staff        4096 Mar 12 05:58 ..
-rw-r--r--    1 usuario   usuarios      266 Mar 12 05:58 .alias
-rw-r--r--    1 usuario   usuarios      509 Mar 12 05:58 .bash_profile
-rw-r--r--    1 usuario   usuarios     1093 Mar 12 05:58 .bashrc
-rw-r--r--    1 usuario   usuarios      375 Mar 12 05:58 .cshrc
```

Las herramientas de que se disponen para modificar los permisos y dueños de los ficheros son: `chown, chgrp, chmod`.

5.- Gestión de software

Los programas, aplicaciones o componentes de las distribuciones de Linux se distribuyen en forma de paquetes. Un paquete es un fichero comprimido que contiene el programa en sí mismo, metainformación del programa y puede contener elementos auxiliares como documentación.

Las distribuciones tienen dos maneras fundamentales de empaquetar software: mediante ficheros .rpm (Red Hat, Fedora, Suse…) o bien mediante ficheros .deb (Debian, Ubuntu y derivados).

Los ficheros .deb pueden instalarse mediante el comando dpkg:

```
user@host:~# dpkg –i /ruta_al_fichero/nombre_fichero.deb
```

Es decir, una vez de que disponemos del fichero .deb le indicamos al programa dónde se encuentra y ordenamos su instalación con la opción –i (install). Si quisiéramos desinstalar un programa usaríamos la opción –r (remove), eso sí pasando como argumento el nombre del programa no el nombre del fichero .deb (y, por tanto, sin ruta alguna):

```
user@host:~# dpkg –r nombre_programa
```

Dpkg es un gestor de software de bajo nivel. Esto quiere decir que si el programa que estamos instalando depende de otros programas que no están instalados (por ejemplo de alguna librería externa), simplemente dará error y le corresponderá al usuario buscar la librería e instalarla por su cuenta.

Para solventar esto y hacer que la gestión de software sea más sencilla disponemos de la suite de herramientas APT (Advanced Package Tool). Apt usa un fichero de configuración (`/etc/apt/sources.list`) donde se detallan URL's a repositorios que albergan los paquetes de software y una base de datos con los paquetes disponibles, los instalados y los instalables.

Mediante el comanado `apt-update` se actualiza la base de datos de software disponible y se revisa si existen versiones actualizadas de software ya instalado.

Mediante el comando `apt-upgrade` se actualiza el sistema. Se instalan las versiones nuevas de aquellos paquetes que estén instalados.

Además se dispone del comando `apt-get` que permite instalar y desinstalar programas:

```
user@host:~# apt-get install nombre_programa
```

Instala el programa indicado. Apt-get accede al repositorio, descarga el paquete y lo instala. Si el software tuviera alguna dependencia descargaría previamente dichas dependencias e instalaría y configuraría todo lo necesario antes de instalar el programa solicitado.

```
user@host:~# apt-get remove nombre_programa
```

Desinstala el programa indicado.

Finalmente cabe destacar el programa `apt-cache` que permite buscar un término dentro de la base de datos de software:

```
user@host:~# apt-cache search php
```

Esta orden buscaría aquellos paquetes relacionados con el lenguaje de programación PHP, por ejemplo.

6.- Administración

Todas las tareas de administración del sistema se pueden realizar en modo local desde la consola del equipo, pero también desde el interfaz gráfico o incluso de forma

remota mediante ssh o herramientas web. En esta sección se analiza la administración local desde consola.

6.1.- Arranque y apagado del sistema

Se pueden configurar los servicios que se lanzan en el arranque del sistema, indicando también cuándo y cómo hay que pararlos. Debian arranca ejecutando el programa *init*. El archivo de configuración de *init* es */etc/inittab*. La entrada initdefault determina el nivel de ejecución inicial del sistema. También se puede obtener el nivel de ejecución actual mediante el comando *runlevel*.

Los primeros scripts que se ejecutan a continuación son los que se encuentran en el directorio */etc/rcS*. Estos scripts son los encargados de montar el sistema de ficheros *root* y */proc*, eliminar temporales y archivos de bloqueo, establece el reloj, activar la partición de intercambio, iniciar las interfaces de red, activar el teclado, cargar los módulos, cargar las variables de entorno, revisar y montar los sistemas de ficheros, activar las quotas, inicializar los puertos USB, etc. Los elementos que se encuentran en el directorio */etc/rcS* son enlaces simbólicos a los scripts que se encuentran en el directorio /etc/init.d

A continuación se ejecutan los scripts de inicialización de los servicios del nivel de ejecución por defecto. Estos scripts se encuentran en los directorios */etc/rcX* donde *X* es el nivel de ejecución. Por ejemplo, /etc/rc5.d/ corresponde al nivel de ejecución número 5. Los niveles de ejecución *(run levels*) definen los diferentes estados de funcionamiento del sistema:

0 Parada del sistema
1 Modo monousuario
2 Modo multiusuario
3 Modo multiusuario
4 No usado
5 Modo multiusuario con interfaz gráfico
6 Parada y arranque
7-9 No se usan

Los script que comienzan por la letra *S* ponen en marcha servicios, mientras que aquellos que comienzan por la letra *K* los finalizan. El script */etc/init.d/rc* procesa todos los archivos K y S de los directorios */etc/rcX.d*

- Para (stop) aquellos procesos que comienzan por K (kill).
- Lanza (start) los que comienzan por S.
- Después de la letra S o K hay dos dígitos numéricos que indican el orden de ejecución. El orden es ASCII.
- Todos los ficheros K o S son enlaces simbólicos a los scrips de cada servicio que están en el directorio /etc/init.d

Para eliminar un servicio en un determinado nivel se puede borrar el vínculo simbólico en */etc/rcn.d/*, pero es más recomendable renombrarlo con algo que no empiece

con S o K y dejarlo por si queremos luego activarlo. Lo que no hay que hacer nunca es eliminar el archivo original en */etc/init.d/*.

Para restablecer el enlace simbólico para que podamos iniciar el servicio usamos la instrucción *update-rc.d*. Por ejemplo, *update-rc.d gdm defaults* permite crear los enlaces simbólicos que ejecutan el script de *gdm* (interfaz gráfico).

6.2.- Gestión de usuarios, grupos y cuentas del sistema

Aunque la gestión de usuarios se puede llevar a cabo desde el interfaz gráfico de un modo más amigable, siempre se pueden emplear los comandos del sistema para esta tarea. A continuación se presentan someramente algunos de ellos.

Crear usuario:

```
user@host:~# useradd usuario
```

Asignar contraseña al usuario:

```
user@host:~# passwd contraseña
```

Crear directorio de usuario:

```
user@host:~# mkdir /home/usuario
```

Asignar propietario y grupo:

```
user@host:~# chown usuario:usuario -R /home/usuario
```

Asignar permisos:

```
user@host:~# chmod 755 –R /home/usuario
```

Asignarle el directorio de usuario al nuevo usuario:

```
user@host:~# usermod –d /home/usuario usuario
```

6.3.- Comprobación del funcionamiento del equipo

A la hora de realizar cualquier modificación en un servicio, es una práctica habitual tener abierto el fichero syslog de forma continua. Esto puede realizarse de muchas maneras, por ejemplo con la instrucción: *tail -f /var/log/syslog.*

El nivel de detalle del registro se denomina prioridad y es configurable. Los valores de prioridad de menor a mayor son:

1. debug
2. info
3. notice
4. warning (warn)

5. error (err)
6. crit
7. alert
8. panic (emerg)

Si configuramos una aplicación con la prioridad *debug*, aparecerá una gran cantidad de apuntes, ya que se utilizan para depurar el funcionamiento de una aplicación. Por el contrario, si configuramos una aplicación con la prioridad *panic* sólo se producirán registros en casos de emergencia.

Edita el fichero de configuración del servidor echo, busca la directiva donde se determina la prioridad de log de ese servicio y modifícala a "debug". Reinicia despué el servicio y monitoriza el fichero */var/log/auth.log*. Ejecuta el comando *echo* y comprueba su impacto en el registro. Al finalizar, devuelve la prioridad al nivel que estaba y reinicia el servicio.

6.4.- Gestión de recursos del sistema

El consumo de recursos críticos como son el uso de CPU y de memoria se lleva a cabo con los comandos *ps* y *top* ya explicados en la primear práctica.

Otros recursos de comunicación entre procesos que se analizarán y desarrollarán en prácticas posteriores pueden ser monitorizados y gestionados con el comando *ipcs*.

6.5.- Gestión de los sistemas de ficheros

El fichero */etc/fstab* lista todos los discos y particiones disponibles, así como las conexiones a sistemas de ficheros remotos. Indica los puntos y modos de montaje de dichos sistemas de ficheros Los campos que ofrece */etc/fstab* son los siguientes:

- **<file systems>** - defines the storage device (i.e. /dev/sda1).

- **<dir>** - tells the mount command where it should mount the <file system> to.

- **<type>** - defines the file system type of the device or partition to be mounted. Many different file systems are supported. Some examples are: ext2, ext3, reiserfs, xfs, jfs, smbfs, iso9660, vfat, ntfs, swap, and auto. The 'auto' type lets the mount command to attempt to guess what type of file system is used, this is useful for removable devices such as CDs and DVDs.

- **<options>** - define particular options for filesystems. Some options relate only to the filesystem itself. Some of the more common options are:

  - **auto** - file system will mount automatically at boot, or when the command 'mount -a' is issued.

- **noauto** - the filesystem is mounted only when you tell it to.
- **exec** - allow the execution binaries that are on that partition (default).
- **noexec** - do not allow binaries to be executed on the filesystem.
- **ro** - mount the filesystem read only.
- **rw** - mount the filesystem read-write.
- **sync** - I/O should be done synchronously.
- **async** - I/O should be done asynchronously.
- **flush** - specific option for FAT to flush data more often, thus making copy dialogs or progress bars to stays up until things are on the disk.
- **user** - permit any user to mount the filesystem (implies noexec,nosuid,nodev unless overridden).
- **nouser** - only allow root to mount the filesystem (default).
- **defaults** - default mount settings (equivalent to rw,suid,dev,exec,auto,nouser,async).
- **suid** - allow the operation of suid, and sgid bits. They are mostly used to allow users on a computer system to execute binary executables with temporarily elevated privileges in order to perform a specific task.
- **nosuid** - block the operation of suid, and sgid bits.
- **noatime** - do not update inode access times on the filesystem. Can help performance.
- **nodiratime** - do not update directory inode access times on the filesystem. Can help performance.
- **relatime** - update inode access times relative to modify or change time. Access time is only updated if the previous access time was earlier than the current modify or change time (similar to noatime, but doesn't break mutt or other applications that need to know if a file has been read since the last time it was modified). Can help performance.

◎ **<dump>** - is used by the dump utility to decide when to make a backup. When installed, dump checks the entry and uses the number to decide if a file system should be backed up. Possible entries are 0 and 1. If 0, dump will ignore the file system, if 1, dump will make a backup. Most users will not have dump installed, so they should put 0 for the <dump> entry.

◎ **<pass>** fsck reads the <pass> number and determines in which order the file systems should be checked. Possible entries are 0, 1, and 2. The root file system should have the highest priority, 1, all other file systems you want to have checked should get a 2. File systems with a <pass> value 0 will not be checked by the fsck utility.

El fichero */etc/mtab* (contracción de m ounted file systems tab le) es un archivo de información del sistemaque muestra los sistemas de ficheros montados actualmente, junto con sus opciones de inicialización. Tiene mucho en común con *fstab*, la diferencia principal es que inicia con las últimas listas de todos los sistemas de ficheros disponibles, mientras que las listas anteriores sólo se montan una vez. Por lo tanto *mtab* es por lo

general en un formato similar a la de *fstab*. Este archivo se encarga de los dispositivos montados y se actualiza automáticamente mediante el comando *mount*.

6.6.- Gestión de las cuotas del sistema

En un servidor con distintos usuarios es fundamental limitar el uso de los sistemas de ficheros, y en concreto el espacio empleado por cada usuario. Esto se logra con el comando *quota*. Para su empleo, es preciso preparar primero la partición en la que queremos poner cuotas editando */etc/fstab* (añadimos lo que está en negrita):

*/dev/hda2/ /home ext3 defaults,***usrquota** *0 2*

Con eso cuando reiniciemos tendremos la partición preparada. Para activarlo ahora sin reiniciar (hasta que volvamos a reiniciar):

mount -o remount,usrquota /home

Creamos dos archivos en la partición en la que queremos crear las cuotas de usuario:

touch /home/aquota.user
touch /home/aquota.group
chmod 600 /home/aquota.user /home/aquota.group

Ahora instalamos quota y quotatools:

apt-get install quota quotatool

Activamos las cuotas:

quotacheck –vagumf

Por último asignamos una cuota de 100 Mb para el usuario "prueba":

quotatool -u prueba -bq 300M -l '100 Mb' /home

Para ver las cuotas asignadas:

repquota /home

Para eliminar la cuota del usuario "prueba":

quotatool -u prueba -bq 0M -l '0 Mb' /home

7.- Automatización de tareas

La automatización de tareas se puede realizer mediante los commandos CRON y AT que se describen a continuación.

7.1.- CRON

Se trata de unos de los servicios básicos de los sistemas Linux. El demonio *cron* siempre está arrancado. La función básica de *cron* es la de ejecutar tareas programadas para un determinado momento, y por un usuario con los privilegios necesarios para poder programarlas. Los ficheros más importantes implicados en el funcionamiento de servicio "cron" son:

- el propio demonio de funcionamiento: crond
- el fichero de configuración (disponible para *root*): /etc/crontab
- el fichero de inicio y parada del demonio: /etc/init.d/cron
- la orden para la programación de tareas (disponible para los usuarios con suficientes privilegios): crontab
- el sistema de informes (*logs*) típico de los sistemas GNU/Linux: /var/log/cron

Para arrancar o parar el demonio cron se deberían ejecutar las órdenes correspondientes:

- Parada del demonio cron: # /etc/init.d/cron stop
- Arranque del demonio cron: # /etc/init.d/cron start

El fichero de configuración (/etc/crontab) está estructurado por líneas, cada una de las cuales contiene una tarea programada, según el siguiente formato:

 # minuto hora dia mes dia_semana usuario orden_a_ejecutar

Evidentemente, cada uno de estos campos tiene un rango de utilización, dependiendo de su naturaleza:

- El campo minuto puede ser definido entre [0-59].
- El campo hora puede ser definido entre [00-23].
- El día del mes, entre [1-31].
- El mes del año, entre los valores [1-12].
- El día de la semana, entre [0-7], asumiendo el 0 como inicio de la semana en domingo, y el valor 7 también como domingo (el 1 será el lunes, 2 martes...).
- El campo usuario es aquel usuario del sistema con permisos que ha definido la tarea programada.
- Orden (o el script) en el que se ejecutará.

Se dispone de símbolos especiales para los cinco primeros campos, que indicarían aspectos genéricos (no un número concreto):

* : indica cualquier valor
, : actúa como separador de una lista de valores
: indica que lo que acompaña es un comentario (no se ejecutará)

- : sirve para indicar un rango de valores
/ : sirve para indicar un paso de valor (por ejemplo, en el campo mes si se indica */3 se está detallando que la tarea se realizará cada tres meses).

Una vez vista su sintaxis, si se deseara ejecutar a las 10 y a las 17 horas, todos los días laborables, la orden `echo "Viva GNU y Linux!"`, se escribiría en el fichero de configuración la línea:

*0 10,17 * * 1-5 echo "Viva GNU y Linux!" | wall*

El día de la semana se puede indicar en dos campos distintos. En caso que los dos forzasen un valor (es decir, que alguno de ellos o los dos no fuesen *), el sistema ejecutará el comando en cualquier de los dos casos (intentará que se cumplan los dos campos). Por ejemplo, en el siguiente ejemplo: 0,45 * 13 * 2 echo " Hola martes o 13!" | Wall. Esta orden se ejecutará cada 45 minutos, todos los martes, y además todos los 13 de cada mes. Además, en los cinco primeros campos se puede optar por los siguientes cadenas:

@reboot: Se ejecuta al iniciarse la máquina.
@yearly: Se ejecuta una vez al año.
@monthly: Se ejecuta una vez al mes.
@weekly: Se ejecuta una vez por semana.
@daily: Se ejecuta una vez al día.
@hourly: Se ejecuta una vez por hora.

Y, por último, también es interesante conocer que cada una de las tareas programadas se ejecuta mediante un shell (/bin/sh), y que están disponibles algunas variables de entorno, como pueden ser LOGNAME, SHELL o NAME.

Ejemplos de tareas programadas en crontab

1. Script que se ejecute **todos los viernes a la 1.00 AM**.
0 1 * * 5 /bin/ejecuta/este/script.sh
Se ejecutaría en la hora 1 minutos 0, todos los viernes del mes ya que en las opciones de mes hemos puesto *.

2. Script que se ejecute **de lunes a viernes a la 1.00AM**.
0 1 * * 1-5 /bin/ejecuta/este/script.sh
Parecido al anterior solo que en el último * ponemos 1-5 para que lo haga de lunes a viernes en vez de 1 que era solo para el viernes.

3. Script que se ejecute **cada 10 minutos**.
0,10,20,30,40,50 * * * * /bin/ejecuta/este/script.sh
o lo que sería lo mismo:
*/10 * * * * /bin/ejecuta/este/script.sh

2.2.- AT

Como se ha comentado, el servicio ofertado por "cron" se ejecuta de forma periódica, de modo que las tareas programadas siempre serán realizadas siguiendo los criterios de temporalización programados.

A diferencia de "cron", las tareas que son encomendadas a "at" (y su demonio, "atd") sólo se realizarán una vez. Es decir, la utilidad "at" se utiliza para programar una tarea que se llevará a cabo en un momento determinado, y no se volverá a ejecutar.

Las utilidades que se pueden encontrar directamente relacionadas con "at" son:

- at: orden que se utiliza para añadir nuevas tareas,
- atd: es el demonio responsable de ejecutar las tareas programadas desde "at"
- atq: muestra la lista de tareas pendientes a ejecutar (por el usuario que llama al comando)
- atrm: elimina una tarea de la lista de pendientes.

Al igual que en el caso de "cron", los ficheros relacionados con "at" son:

- /etc/at.allow: si existe, sólo los usuarios listados en este fichero tienen permiso para ejecutar la utilidad.
- /etc/at.deny: si existe este fichero, y no el anterior, los usuarios detallados en él no pueden ejecutar at (ni atrm, ni atq).
- Si no existiesen ninguno de los anteriores ficheros, sólo el usuario *root* puede disponer de la utilidad "at".

Normalmente, después de la instalación se dispone del fichero /etc/at.deny, aunque vacío, de modo que todos los usuarios tienen acceso a la orden "at".

La sintaxis que se utiliza es la orden (at) seguido de los parámetros:

- HH[:]MM[am|pm] [Mes día]
- Se puede añadir a la fecha/hora un número (seguido de "minutes","hours","days","weeks")
- También se pueden añadir valores relativos como "now", "midnight", "noon", "teatime", "today" o "tomorrow".

Ejemplos de utilización:

- $ at 10am tomorrow
- $ at 10am Jun 30
- $ at 1730 Feb 28 + 3 days
- $at 12am tomorrow < copia.sh

Cuestiones a resolver

1.- Programar una tarea que recuerde diez minutos antes de la finalización de la sesión de prácticas (13:50 h) la necesidad de entregar la práctica en la herramienta MiAulario mediante el volcado en pantalla de un mensaje de aviso.

2.- Seleccione el nivel de arranque con el que desee iniciar la sesión de prácticas y configure los servicios que han de iniciarse y pararse y el orden en el que deberán hacerlo.

3.- Revise la ocupación de los sistemas de ficheros y fije una cuota de 100 MB para sus usuarios.

4.- Analice los dispositivos montados en el sistema y sus puntos de montaje.