Sistemas Operativos Avanzados
Práctica 3: Llamadas al sistema y programación de una μshell

UAH, Departamento de Automática, ATC-SOL
`http://atc1.aut.uah.es`

Temas 1-5

Resumen

El objetivo de esta práctica es estudiar las llamadas al sistema relacionadas con archivos y practicar su uso programando una pequeña *shell* (intérprete de órdenes).

1. Introducción

El objetivo de esta práctica es estudiar algunas llamadas al sistema y practicar su uso. El el apartado 2 se describen las llamadas al sistema básicas para el acceso a archivos, y se propone la realización de un programa sencillo que las use. En el apartado 3 se propone la realización de una pequeña *shell* (intérprete de órdenes), que llamaremos μshell, y que requiere el uso de otras llamadas al sistema.

2. Llamadas al sistema básicas para manejo de archivos

2.1. Abrir un archivo: `open()`

La llamada al sistema `open()` es el primer paso que debe ejecutar todo proceso que quiera acceder a los datos de un archivo. Permite solicitar porsteriormente operaciones de lectura/escritura en el archivo abierto. Su sintaxis es:

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

Parámetros:

- `pathname`: nombre del archivo (precedido por la ruta opcionalmente).
- `flags`: modo de apertura (lectura, escritura, etc.).
- `modes`: permisos del archivo en caso de que se deba crear.

Valor devuelto: entero que representa el descriptor del archivo; mayor o igual que 0 si la ejecución ha sido correcta. El descriptor de archivo será utilizado cada vez que se desee realizar una operación sobre el archivo.

2.1.1. Pruebas recomendadas para `open()`:

Consulte el manual[1,2] y programe los experimentos que considere necesarios hasta tener claro el comportamiento y el valor devuelto por `open()` en los siguientes casos:

1. Se abre un archivo que existe y otro que no existe.
2. Se abre un archivo con permisos y sin permisos.
3. Se realizan varias llamadas a open consecutivas sobre distintos archivos y sobre el mismo archivo con el mismo modo de apertura.
4. Dos programas diferentes realizan `open()` sobre distintos archivos y sobre el mismo archivo. Use varias terminales, o ejecución en segundo plano (&) para comprobar si los programas pueden acceder a la vez a un mismo archivo.

2.2. Leer el contenido de un archivo: `read()`

La llamada al sistema `read()` permite acceder a los datos almacenados como contenido de un archivo guardándolos en posiciones de memoria (variables). Su sintaxis es:

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

Parámetros:

- `fd`: descriptor del archivo (devuelto por una llamada a `open()`) sobre el que se va a realizar la lectura.
- `buffer`: dirección de memoria donde se van a colocar los datos leídos.
- `count`: número de bytes que el usuario quiere leer.

Valor devuelto: entero que indica el número de bytes leídos; mayor o igual que 0 si la ejecución ha sido correcta.

El número de bytes leídos permite controlar cuándo se ha alcanzado el final del archivo. No existe una marca específica de Fin de Archivo. Las lecturas realizadas cuando el puntero del archivo está al final de éste, devolverán cero bytes leídos.

2.2.1. Pruebas recomendadas para `read()`:

Consulte el manual[3] y programe los experimentos que considere necesarios hasta tener claro el comportamiento y el valor devuelto por `read()` en los siguientes casos:

5. Leer de un archivo un número de bytes menor y mayor de los que contiene.
6. Leer de un archivo cuando ya se han leído todos los datos contenidos (equivalente a leer de un archivo vacío).
7. Leer de un archivo utilizando dos descriptores distintos (dos `open()` sobre el mismo archivo).

[1] Las llamadas al sistema están en la sección 2 del manual. En la línea de órdenes, teclee: `man 2 open`
[2] Conviene tener instalados los siguientes paquetes: `manpages-dev glibc-doc build-essential`
[3] En la línea de órdenes, teclee: `man 2 read`

2.3. Escribir en el contenido de un archivo: `write()`

La llamada al sistema `write()` permite modificar los datos, o añadir más, almacenados como contenido de un archivo a partir de información almacenada en posiciones de memoria (variables). Su sintaxis es:

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

Parámetros:

- `fd`: descriptor del archivo (devuelto por una llamada a `open()`) sobre el que se va a realizar la escritura.
- `buffer`: dirección de memoria de donde se van a tomar los datos a escribir.
- `count`: número de bytes que el usuario quiere escribir.

Valor devuelto: entero que indica el número de bytes escritos; mayor o igual que 0 si la ejecución ha sido correcta.

2.3.1. Pruebas recomendadas para `write()`:

Consulte el manual[4] y programe los experimentos que considere necesarios hasta tener claro el comportamiento y el valor devuelto por `write()` en los siguientes casos:

8. Escribir en un archivo un número de bytes menor y mayor de los que contiene.
9. Escribir en un archivo cuando se está al final del archivo (equivalente a escribir en un archivo vacío).
10. Escribir en un archivo utilizando dos descriptores distintos (dos `open()` sobre el mismo archivo).

2.4. Reposicionar el puntero de lectura/escritura de un archivo: `lseek()`

La llamada al sistema `lseek()` permite modificar la posición donde se va a realizar la siguiente operación de lectura o escritura sobre un archivo ya abierto, es decir, modifica el valor del puntero que se almacena en la entrada correspondiente de la tabla de archivos abiertos. Su sintaxis es:

```
off_t lseek (int fd, off_t offset, int whence)
```

Parámetros:

- `fd`: descriptor del archivo (devuelto por una llamada a `open()`) al que se le va a modificar el puntero.
- `offset`: número de bytes a desplazar el puntero. Puede ser positivo o negativo. El significado exacto de este parámetro depende del valor del parámetro `whence`.

[4] En la línea de órdenes, teclee: `man 2 write`

- `whence`: posición desde la que se va a sumar el desplazamiento indicado en el parámetro anterior. Puede ser SEEK_SET (principio del archivo), SEEK_CUR (posición actual del puntero) o SEEK_END (final del archivo).

Valor devuelto: posición resultante en bytes desde el principio del archivo (valor del puntero), ó -1 en caso de producirse un error.

Es posible situar el puntero más allá del final de un archivo. En caso de realizar una escritura en esa posición, el hueco intermedio no escrito se rellenará con ceros.

2.4.1. Pruebas recomendadas para `lseek()`:

Consulte el manual[5] y programe los experimentos que considere necesarios hasta tener claro el comportamiento y el valor devuelto por `lseek()` en los siguientes casos:

11. Saltar al final del archivo y escribir datos.
12. Saltar al principio del archivo y leer datos.
13. Posicionarse en un lugar mayor del tamaño del archivo y escribir datos.
14. Averiguar la posición actual con un salto a {`offset`=0, `whence`=SEEK_CUR}.
15. Averiguar el tamaño del archivo.

2.5. Proyectar en memoria el contenido de un archivo: `mmap()`

La llamada al sistema `mmap()` permite proyectar el contenido de un archivo en posiciones de memoria, pudiendo, tras la proyección, acceder al contenido del archivo mediante variables que apunten a la zona proyectada. Esto significa que, tras llamar a `mmap()`, se puede escribir o leer el contenido de un archivo utilizando las posiciones correspondientes del array que comienza en la dirección donde se ha proyectado. La sintaxis de `mmap()` es:

```
void *mmap (void *start, size_t length, int prot,
            int flags, int fd, off_t offset)
```

Parámetros:

- `start`: dirección de memoria sugerida para realizar la proyección. Habitualmente se especifica la dirección 0 (NULL), que le indica al sistema operativo que busque y asigne una dirección válida.
- `length`: número de bytes a proyectar.
- `prot`: protección de la memoria deseada. Puede ser PROT_EXEC (las páginas van a ser ejecutadas), PROT_READ (las páginas van a ser leídas), PROT_WRITE (las páginas van a ser escritas) o PROT_NONE (se reservará espacio virtual, pero las páginas no serán accesibles hasta que se cambie la protección con `mprotect()`). Estos valores se pueden combinar con la operación *OR* bit a bit (operador `|` del lenguaje `C`)[6].

[5] En la línea de órdenes, teclee: `man 2 lseek`
[6] Por ejemplo: `PROT_READ|PROT_WRITE`

- `flags`: opciones de proyección. Puede ser MAP_FIXED (no utilizar una dirección diferente a la pasada), MAP_SHARED (la zona proyectada puede ser compartida con otros procesos) o MAP_PRIVATE (la zona de memoria es privada). Es necesario especificar exactamente una de las dos últimas opciones. Adicionalmente, se puede especificar MAP_FIXED, combinándola con MAP_SHARED o MAP_PRIVATE mediante una operación *OR* bit a bit[7].
- `fd`: descriptor del archivo (devuelto por una llamada a `open()`) que va a ser proyectado.
- `offset`: posición del archivo desde la que se va a realizar la proyección. Debe ser un múltiplo del tamaño de página[8].

Valor devuelto: posición de memoria donde se ha realizado la proyección en caso de que se haya efectuado correctamente, o MAP_FAILED[9] en caso de producirse un error o si no se ha podido asignar la zona de memoria solicitada.

El acceso a archivos mediante proyección en memoria es más rápido que el realizado por las llamadas de lectura y escritura por varios motivos. En primer lugar, con el acceso mediante proyección no es necesario acceder a las tablas intermedias en cada operación. En segundo lugar, los accesos fuera del espacio de memoria de un proceso también son más lentos. Finalmente, los accesos con `read()` y `write()` requieren una llamada al sistema por cada operación, mientras que con proyección en memoria, gracias a la localidad espacial y temporal, la mayoría de las lecturas/escrituras no requerirán ninguna acción inmediata por parte del sistema operativo.

El tiempo que se tarda en proyectar una parte de un archivo es el mismo que se tarda en proyectar el archivo completo, ya que el sistema operativo sólo almacena la relación entre las posiciones de memoria virtual y las de disco, dejando las operaciones de lectura o escritura para los momentos en que realmente se realizan. Por esto, es frecuente que cuando se van a realizar varias operaciones sobre un archivo, se proyecte de forma completa, para lo cual es necesario conocer antes el tamaño de éste.

2.5.1. Pruebas recomendadas para `mmap()`:

Consulte el manual[10] y programe los experimentos que considere necesarios hasta tener claro el comportamiento y el valor devuelto por `mmap()` en los siguientes casos:

16. Leer datos del contenido del archivo.

2.6. Liberar la memoria donde está proyectado un archivo: `munmap()`

La llamada al sistema `munmap()` permite liberar las direcciones de memoria asociadas a un archivo previamente proyectado. Su sintaxis es:

```
int munmap(void *start, size_t length)
```

[7] Por ejemplo: `MAP_SHARED|MAP_FIXED`
[8] El tamaño de página se puede obtener con `sysconf(_SC_PAGE_SIZE)`
[9] MAP_FAILED está definido como `(void*)-1`
[10] En la línea de órdenes, teclee: `man 2 mmap`

Parámetros:

- `start`: dirección de memoria a liberar. Debe coincidir con un valor devuelto por una llamada previa a `mmap()`.
- `length`: número de bytes a liberar.

Valor devuelto: entero que indica si la operación se ha realizado de forma correcta (0) o incorrecta (-1).

Después de liberar una zona de memoria previamente proyectada, todos los accesos posteriores a dicha zona producirán referencias a memoria inválidas.

2.6.1. Pruebas recomendadas para `munmap()`:

Consulte el manual[11] y programe los experimentos que considere necesarios hasta tener claro el comportamiento y el valor devuelto por `munmap()` en los siguientes casos:

17. Leer datos del contenido del archivo después de llamar a `munmap()`.

2.7. Cerrar un archivo: `close()`

La llamada al sistema `close()` deja de nuevo disponible un descriptor previamente abierto, liberando así mismo toda la información que el sistema operativo almacena relacionada con dicho archivo abierto. Cuando un proceso no va a utilizar más un archivo que tiene abierto, éste debe ser cerrado para asegurar que todas las operaciones con dicho archivo se completan (el sistema de *buffering* que utilizan los sistemas UNIX, hace que una operación no ocurra justo en el instante en que se realiza la llamada al sistema, sino en el momento en que sea más conveniente para el rendimiento del sistema). La sintaxis de esta llamada es:

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

Parámetros:

- `fd`: descriptor de archivo (devuelto por una llamada a `open()`) que va a ser cerrado.

Valor devuelto: entero que indica si la operación se ha realizado de forma correcta (0) o incorrecta (-1).

Al cerrar un archivo se libera la entrada correspondiente en la tabla de descriptores de archivos y, si no hay más entradas de otros procesos que apunten a la misma entrada de la tabla de archivos abiertos, también se libera esta última entrada.

2.7.1. Pruebas recomendadas para `close()`:

Consulte el manual[12] y programe los experimentos que considere necesarios hasta tener claro el comportamiento y el valor devuelto por `close()` en los siguientes casos:

18. Cerrar un descriptor no existente.
19. Leer o escribir datos en un archivo después de cerrarlo.

[11] En la línea de órdenes, teclee: `man 2 munmap`
[12] En la línea de órdenes, teclee: `man 2 close`

2.8. Contar, con llamadas al sistema, las apariciones de un carácter en un archivo

Siguiendo las instrucciones que se detallan a continuación, realice un programa que cuente las apariciones de un determinado carácter dentro de un archivo.

El programa debe estar compuesto de tres módulos en lenguaje `C`, uno que contendrá el programa principal, la comprobación de argumentos y la visualización del resultado, un segundo módulo que contendrá una función que devolverá el carácter existente en cierta posición de un archivo ya abierto y un tercer módulo que devolverá el número de apariciones de un carácter dentro de un archivo ya abierto.

Estos dos últimos módulos no se invocarán el uno al otro, sino que proporcionarán métodos alternativos de acceso a los datos del archivo. La función que lee un carácter debe ser implementada sin utilizar proyección en memoria y la que cuenta el número de apariciones debe ser implementada utilizando proyección en memoria.

Será necesario también crear un archivo `Makefile` que facilite la compilación de los distintos módulos.

Los pasos para realizar el programa se detallan a continuación:

1. Cree un archivo `Makefile` que genere un programa ejecutable `contar` a partir de los archivos objeto correspondientes a los módulos fuente `principal.c`, `leercar_R.c` y `contar_M.c`. También existirán sendos archivos `leercar_R.h` y `contar_M.h` que contendrán la declaración de las funciones correspondientes. Estas funciones serán invocadas desde `principal.c`. Recuerde que todas las compilaciones deben realizarse con el modificador `-Wall`. Los archivos objeto de las dos funciones implementadas se almacenarán en una biblioteca denominada `libcontar.a` mediante la orden `ar`[13]. Consulte la página *man* de dicha orden para saber cómo utilizarla. La figura 1 muestra un esquema del proceso de compilación y enlazado que deberá especificar el archivo `Makefile`.

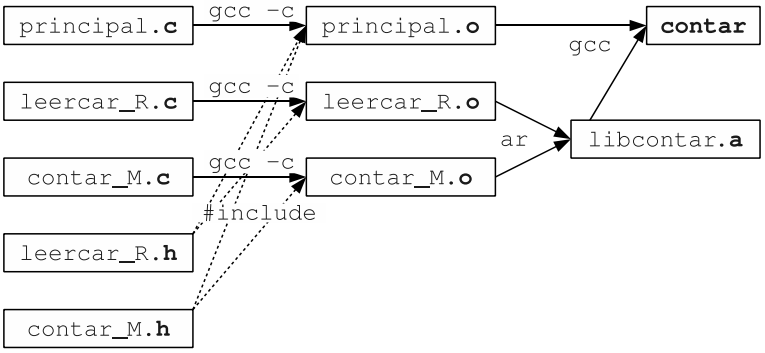

Figura 1: Diagrama de compilación y enlazado del programa `contar`

[13]Ejemplo de uso de la orden `ar` y del archivo `.a` resultante:

```
user@host:$ ar -r libcolores.a rojo.o azul.o amarillo.o
user@host:$ gcc -o programa principal.o -L . -l colores
```

2. Cree los archivos de cabecera `leercar_R.h` y `contar_M.h` con las siguientes declaraciones respectivamente:

```
char LeerCaracter (int fd, int posicion);
int ContarCaracteres (int fd, char caracter);
```

3. Cree los módulos `leercar_R.c` y `contar_M.c`, que contendrán la definición de la funciones declaradas en los archivos de cabecera correspondientes. La función `LeerCaracter()` debe devolver, sin utilizar proyección en memoria, el carácter ubicado en la posición indicada de un archivo del que recibe su descriptor. La función `ContarCaracteres()` debe calcular, utilizando proyección en memoria, el número de ocurrencias del carácter indicado localizadas en el archivo del que recibe su descriptor.

4. Cree el módulo `principal.c`, que comprobará que el número de argumentos recibidos es correcto y abrirá el archivo pasado como argumento. Después de esto, llamará a la función correspondiente y mostrará por pantalla el resultado devuelto por ésta. La sintaxis de la llamada al ejecutable es la siguiente:

   Para contar leyendo con `read()`:
   `user@host:$ ./contar R` *archivo carácter*

   Para contar con proyección en memoria:
   `user@host:$ ./contar M` *archivo carácter*

   Por ejemplo:

```
user@host:$ ./contar R archivo a
La letra a aparece 43 veces
user@host:$ ./contar M archivo a
La letra a aparece 43 veces
```

Compruebe que la ejecución del programa es correcta tras la generación del ejecutable (recuerde que ésta no debe producir ningún aviso, ni en la compilación ni en el enlazado, utilizando el modificador `-Wall`) y depúrelo si fuera necesario.

3. Programación de una μshell

3.1. Base teórica: qué es una *shell* y cómo funciona

Como ya es conocido, un intérprete de órdenes es un programa que solicita palabras clave (órdenes) por la entrada estándar y, en función de lo introducido, realiza unas tareas u otras. El ciclo de ejecución de un intérprete de órdenes (también conocido como *shell*) se puede resumir en cuatro etapas:

1. **Espera de orden de usuario.** El programa, normalmente, muestra un mensaje por pantalla, denominado *prompt*, para invitar al usuario a que teclee una orden.

2. **Procesado de la orden (*parsing*).** En esta etapa se realizan un conjunto de transformaciones sobre la línea introducida por el usuario. Se pueden distinguir cinco tipos, aunque depende de las capacidades de la *shell*que los incluya o no: sustituciones de históricos, alias, sustitución de variables, sustituciones de órdenes y expansión de archivos.

3. **Ejecución de la orden.** En esta etapa es donde se localiza y ejecuta la orden. Los intérpretes pueden clasificar las órdenes en dos tipos:

   - Órdenes internas: son aquellas que ejecuta la propia *shell*. El código que ejecuta la orden está incluido dentro del código de la *shell*.
   - Órdenes externas: son aquellas cuyo ejecutable es un programa externo a la *shell*. Para su ejecución, el intérprete de órdenes se duplica (crea un proceso nuevo mediante una llamada a `fork()`) y, a continuación, el nuevo proceso se recubre con el código del programa externo a ejecutar (mediante una llamada a `exec()` o similar).

4. **Espera de fin de orden.** Si la orden no se ha ejecutado en segundo plano (en *background*, indicándolo mediante '&'), el proceso inicial (la *shell*) debe esperar a la finalización del nuevo proceso antes de volver a la etapa 1.

3.2. Programar una μshell que reconozca algunas órdenes internas de manejo de archivos y directorios

Siguiendo las instrucciones que se detallan a continuación, realice una μshell (un pequeño intérprete de órdenes).

Junto con este enunciado se proporcionan dos archivos de código (`parser.h` y `parser.c`) que facilitan la etapa de procesado de la orden. Estudie el programa `prueba.c` (también facilitado con este enunciado), que ilustra el uso de `parser.h` y `parser.c`. El siguiente listado muestra un ejemplo de compilación y ejecución del programa `prueba.c`:

```
user@host:$ gcc -Wall prueba.c parser.c -o prueba
user@host:$ ./prueba
Introduzca órdenes (pulse Ctrl-D para terminar)
$ mi_programa par\ 1 <in.txt 2>err.txt "par 2" &
        Orden cruda: "mi_programa par\ 1 <in.txt 2>err.txt "par 2" &"
        Número de argumentos: 3
                argv[0]: "mi_programa"
                argv[1]: "par 1"
                argv[2]: "par 2"
                argv[3]: NULL
        Entrada: "in.txt"
        Salida de err.: "err.txt"
        Ejecutar en segundo plano: Sí
$
```

Utilice la función `main()` de `prueba.c` como base para la μshell. Como se muestra en el ejemplo anterior, el *parser* facilitado es capaz de interpretar comillas, redirecciones de entrada/salida, el carácter de escape '\', y el carácter de ejecución en segundo plano '&'. Con el fin de simplificar el ejercicio, ignore las redirecciones y la ejecución en segundo plano. La μshell se limitará a ejecutar las órdenes internas, sin obedecer a las redirecciones y sin necesidad de llamar a `fork()` o `exec()`.

Las órdenes internas a implementar son las siguientes:

1. `mipwd`, que mostrará el directorio actual.
2. `mils`, que debe mostrar un listado de las entradas del directorio que se le pase como parámetro, o del directorio actual si no se especifica ninguno. Además, si se añade el parámetro opcional `-l`, el listado deberá detallar por cada entrada los siguientes datos:
   - UID (identificador de usuario) del propietario
   - GID (identificador de grupo) del propietario
   - permisos de acceso
   - tipo de archivo
   - tamaño
   - fecha de modificación
   - nombre del archivo o directorio
3. `mimkdir`, que creará un nuevo directorio con el nombre que se le pase como parámetro.
4. `mirmdir`, que eliminará el directorio que se le pase como parámetro.
5. `micd`, que cambiará al directorio que se le pase como parámetro o, si no se le pasa ningún parámetro, al directorio donde se lanzó el intérprete de órdenes.
6. `micat`, que mostrará el contenido del archivo que se le pase como parámetro. Esta orden se implementará con proyección en memoria.
7. `micp`, que copiará, utilizando `read()` y `write()`, el contenido del archivo pasado como primer parámetro en el archivo pasado como segundo parámetro.
8. `mirm`, que eliminará (bueno, sólo llamará a `unlink()`) el archivo que se le pase como parámetro.
9. `exit`, que terminará la ejecución de la μshell.

Estructure el programa en varios archivos `.c` y `.h`, y confeccione un `makefile` para agilizar la compilación. Como mínimo tendrá que haber una función independiente por cada orden interna. Se recomienda utilizar funciones adicionales para llevar a cabo tareas que serán necesarias en diferentes puntos del programa[14].

[14] Por ejemplo, la función `main()` necesitará averiguar el directorio actual antes de empezar a interpretar órdenes, mientras que la orden `mipwd` deberá averiguar el directorio actual cada vez que sea ejecutada.

Además de las llamadas al sistema del apartado 2, necesitará utilizar las llamadas al sistema que se describen a continuación:

- Crear un archivo:

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

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

- Borrar[15] un archivo:

```
#include <unistd.h>

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

- Obtener el estado de un archivo en una estructura de tipo `struct stat`:

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

int stat (const char *path, struct stat *buf);
int fstat (int fd, struct stat *buf);
int lstat (const char *path, struct stat *buf);
```

- Abrir un directorio (devuelve una especie de descriptor de directorio):

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

DIR *opendir (const char *name);
```

- Obtener la información de una entrada del directorio cuyo descriptor se pasa como parámetro:

```
#include <dirent.h>

struct dirent *readdir (DIR *dirp);
```

- Cerrar el directorio cuyo descriptor se pasa como parámetro:

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

int closedir (DIR *dirp);
```

[15] En realidad, `unlink()` decrementa el contador de entradas que conducen al archivo, borrando el archivo sólo si la cuenta llega a cero

- Crear un directorio:

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

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

- Borrar un directorio:

```
#include <unistd.h>

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

- Obtener el *path* absoluto del directorio actual:

```
#include <unistd.h>

char *getcwd (char *buf, size_t size);
```

- Cambiar el directorio actual:

```
#include <unistd.h>

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

Para obtener información más detallada consulte el manual en línea (orden `man`). Recuerde que probablemente tendrá que especificar la sección 2 o la sección 3 para evitar que `man` muestre las páginas de secciones anteriores. Por ejemplo, `man mkdir` muestra la página de la orden `mkdir` de la *shell* (sección 1); mientras que `man 2 mkdir` muestra la página de la llamada al sistema `mkdir()` (sección 2).