Práctica 3: Conceptos básicos de sistemas de ficheros

Índice

1 Objetivos **1**

2 Ejercicios **1**

Ejercicio 1: Copia de ficheros regulares 1
Ejercicio 2: Enlaces simbólicos. 2
Ejercicio 3: Desplazamiento del marcador de posición en ficheros. 3
Ejercicio 4: Recorrido de directorios. 3
Ejercicio 5: Administración de ficheros y directorios 4
Ejercicio 6: Permisos y modos de apertura 4

1 Objetivos

En esta práctica vamos a hacer varios ejercicios orientados a afianzar los conceptos básicos de ficheros y directorios en los sistemas POSIX. Trabajaremos las llamadas al sistema: `open`, `read`, `write`, `close`, `lstat`, `readlink`, `symlink`, `lseek`, `opendir` y `readdir`.

Se aconseja al alumno que cree un directorio para la práctica con un subdirectorio por ejercicio. En las instrucciones se asume que el ejercicio N se hace en un subdirectorio llamado ejercicio*N* dentro del directorio común de la práctica.

El archivo ficheros_p3.tar.gz contiene una serie de ficheros que pueden usarse como punto de partida para el desarrollo de los ejercicios de esta práctica, así como unos makefiles que pueden usarse para la compilación de los distintos proyectos.

2 Ejercicios

Ejercicio 1: Copia de ficheros regulares

Diseña un programa copy.c que permita hacer la copia de un fichero regular usando las llamadas al sistema del estándar POSIX: open, read, write y close. Se deben consultar sus páginas de manual, prestando especial atención a los flags de apertura: O_RDONLY, O_WRONLY, O_CREAT, O_TRUNC.

El programa recibira dos parámetros por la línea de llamadas. El primero será el nombre del fichero a copiar (fichero origen) y el segundo será el nombre que queremos darle a la copia (fichero destino).

El programa debe realizar la copia en bloques de 512B, usando un array local como almacenamiento intermedio entre la lectura y la escritura. El programa debe ir leyendo bloques de 512 bytes del fichero origen y escribiendo los bytes leídos en el fichero destino. Debe tenerse en cuenta que si el tamaño del fichero no es múltiplo de 512 bytes la última vez no se leerán 512 bytes, sino lo que quede hasta el final del fichero (conultar el apartado RETURN VALUE en la página de manual de read). Por ello siempre se deben escribir en el fichero destino tantos bytes como se hayan leído del fichero origen.

Para comprobar el efecto de O_TRUNC, se sugiere al alumno que antes de ejecutar su programa de copia, cree un fichero con cualquier contenido que se llame como el fichero destino. Después puede copiar otro fichero usando el nombre elegido para el fichero destino y comprobar que el contenido anterior desaparece al usarse el flag O_TRUNC.

Para comprobar el funcionamiento correcto de nuestro programa podemos usar los comandos de shell diff y hexdump (este último para ficheros binarios).

Ejercicio 2: Enlaces simbólicos.

Lo primero que vamos a hacer en este ejercicio es crear un enlace simbólico a un fichero cualquiera usando el comando `ln`. Por ejemplo, si queremos crear un enlace que se llame *mylink* y que apunte al fichero *../ejercicio1/Makefile* usaremos el siguiente comando del shell:

```
$ ln -s ../ejercicio1/Makefile mylink
```

Invocando `ls -l` podremos comprobar que el fichero creado es realmente un enlace simbólico y veremos el fichero apuntado:

```
$ ls -l
...
lrwxrwxrwx 1 christian christian   22 Jul 14 13:23 mylink -> ../ejercicio1/Makefile
...
```

Ahora usaremos nuestro programa de copia para copiar el enlace simbólico. Asumiendo que dicho programa es *../ejercicio1/copy*, ejecutamos:

```
$ ../ejercicio1/copy mylink mylinkcopy
```

¿Qué tipo de fichero es *mylinkcopy*? ¿Cuál es el contenido del fichero *mylinkcopy*? Se pueden usar los comandos `ls`, `stat`, `cat` y `diff` para obtener las respuestas a estas preguntas.

Es posible que este sea el comportamiento que deseemos, pero también es posible que no. ¿Y si queremos que la copia de un enlace simbólico sea otro enlace simbólico que apunte al mismo fichero que apuntaba el enlace simbólico original?

Vamos a hacer una modificación de nuestro programa de copia del ejercicio anterior, que llamaremos *copy2.c*. Podemos empezar copiando el programa anterior para luego modificarlo. Haremos entonces una copia usando el comando `cp`:

```
$ cp ../ejercicio1/copy.c copy2.c
```

Después editaremos el fichero *copy2.c* de modo que: 1. Antes de hacer la copia identifique si el fichero origen es un fichero regular, un enlace simbólico u otro tipo de fichero, haciendo uso de la llamada al sistema `lstat` (consultar su página de manual).

2. Si el fichero origen es un fichero regular, haremos la copia como en el ejercicio anterior.
3. En cambio, si el fichero origen es un enlace simbólico no tenemos que hacer la copia del fichero apuntado sino crear un enlace simbólico que apunte al mismo fichero al que apunta el fichero origen. Para ello tenemos que seguir los siguientes pasos:
    a. Reservar memoria para hacer una copia de la ruta apuntada. Una llamada a `lstat` sobre el fichero origen nos permitirá conocer el número de bytes que ocupa el enlace simbólico, que se corresponde con el tamaño de esta ruta sin el carácter *null* ('\0') de final de cadena (consultar la página de manual de `lstat`). Por tanto sumaremos uno al tamaño obtenido de `lstat`.
    b. Copiar en este buffer la ruta del fichero apuntado haciendo uso de la llamada al sistema `readlink`. Deberemos añadir manualmente el caracter null de final de cadena.
    c. Usar la llamada al sistema `symlink` para crear un nuevo enlace simbólico que apunte a esta ruta.

Debéis consultar las páginas de manual de `lstat`, `readlink` y `symlink`.

4. Si el fichero origen es de cualquier otro tipo (por ejemplo un directorio) mostrarán un mensaje de error y el programa terminará.

Ejercicio 3: Desplazamiento del marcador de posición en ficheros.

En este ejercicio vamos a crear un programa *mostrar.c* similar al comando `cat`, que reciba como parámetro el nombre de un fichero y lo muestre por la salida estándar. En este caso asumiremos que es un fichero regular. Además, nuestro programa recibirá dos argumentos que parsearemos con `getopt` (consultar su página de manual):

- `-n N`: indica que queremos saltarnos N bytes desde el comienzo del fichero o mostrar únicamente los N últimos bytes del fichero. Que se haga una cosa o la otra depende de la presencia o no de un segundo flag -e. Si el flag -n no aparece N tomará el valor 0.
- `-e`: si aparece, se leerán los últimos N bytes del fichero. Si no aparece, se saltarán los primeros N bytes del fichero.

El programa debe abrir el fichero indicado en la línea de comandos (consultar `optind` en la página de manual de `getopt`) y después situar el marcador de posición en la posición correcta antes de leer. Para ello haremos uso de la llamada al sistema `lseek` (consultar la página de manual). Si el usuario ha usado el flag -e debemos situar el marcador N bytes antes del final del fichero. Si el usuario no ha usado el flag -e debemos avanzar el marcador N bytes desde el comienzo del fichero.

Una vez situado el marcador de posición, debemos leer byte a byte hasta el final de fichero, escribiendo cada byte leído por la salida estándar (descriptor 1).

Ejercicio 4: Recorrido de directorios.

En este ejercicio vamos a crear un programa *espacio.c* que reciba una lista de nombres de fichero como parámtros de la llamada, y calculará para cada uno el número total de kilobytes reservados por el sistema para almacenar dicho fichero. En caso de que alguno de los ficheros procesados sean de tipo directorio, se sumarán también los kilobytes ocupados por los ficheros contenidos el directorio (notar que esto es recursivo, porque un directorio puede contener otros directorios).

Para conocer el número de kilobytes reservados por el sistema para almacenar un fichero podemos hacer uso de la llamada a `lstat`, que nos permite saber el número de bloques de 512 bytes reservados por el sistema.

Para identificar si un fichero es un directorio deberemos hacer una llamada a `lstat` y consultar el campo st_mode (consultar la página de manual de `lstat`).

Para recorrer un directorio, primero deberemos abrirlo usando la función de biblioteca `opendir` y luego leer sus entradas usando la función de biblioteca `readdir`. Consultar las páginas de manual de estas dos funciones. Notar que las entradas de un directorio no tienen un orden establecido y que todo directorio tiene dos entradas "." y "..", que deberemos ignorar si no queremos tener una recursión infinita.

El programa debe mostrar por la salida estándar una línea por fichero de la línea de comandos, con el tamaño total en kilobytes del fichero y el nombre de dicho fichero. Para comprobar si nuestro programa funciona correctamente podemos comparar su salida con la del comando `du -ks`, pasando a este comando la misma lista de ficheros que al nuestro. Notar que se pueden usar los comodines del shell.

Ejemplo de uso:

```
$ ls -l .
total 40
-rwxr-xr-x 1 christian christian 20416 Jul 15 12:41 espacio
-rw-r--r-- 1 christian christian  1639 Jul 15 12:41 espacio.c
-rw-r--r-- 1 christian christian  9056 Jul 15 12:41 espacio.o
```

```
-rw-r--r-- 1 christian christian   273 Jul 15 09:54 Makefile
$ ./espacio .
44K .
$ ./espacio *
20K espacio
4K  espacio.c
12K espacio.o
4K  Makefile
```

Ejercicio 5: Administración de ficheros y directorios

En los dos últimos ejercicios, trabajaremos con los atributos típicos de ficheros, aprendiendo a consultarlos y modificarlos desde línea de comandos, y observando sus implicaciones al interactuar con programas escritos en lenguaje C.

Para ello, se desarrollará un *script* llamado `prepara_ficheros.sh`, que realizará las siguientes acciones preliminares:

1. Creará y accederá a un directorio proporcionado como único argumento al *script*. En caso de no existir, dicho directorio se creará (`mkdir`). En caso de existir, se borrará todo su contenido usando las opciones adecuadas del comando `rm` o mediante el comando `rmdir`.
2. Creará en el directorio especificado un conjunto de ficheros con las siguientes características y nombres:

Nombre	Tipo	Observaciones	Comandos a consultar (man)
subdir	Directorio	–	mkdir
fichero1	Fichero regular	Se creará sin contenido	touch
fichero2	Fichero regular	Se creará y escribirán en él 10 caracteres	echo
enlaceS	Enlace simbólico	Enlace simbólico al fichero `fichero2`	ln
enlaceH	Enlace duro	Enlace duro al fichero `fichero2`	ln

3. Recorrerá todos los ficheros creados, mostrando por pantalla todos sus atributos utilizando la orden `stat`, de la que puede obtenerse más información consultando la página de manual correspondiente (`man 1 stat`).

A tenor de los resultados observados tras la ejecución del *script*, responde razonadamente a las siguientes cuestiones:

1. ¿Cuántos bloques de disco ocupa el fichero `fichero1`? ¿Y el fichero `fichero2`? ¿Cuál es su tamaño en bytes?
2. ¿Cuál es el tamaño reportado para `subdir`? ¿Por qué el campo *número de enlaces (Links)* en este caso es 2?
3. ¿Comparten número de i-nodo alguno de los ficheros o directorios creados? Incluye en tu respuesta el comportamiento ante un directorio, específicamente de las entradas ocultas del directorio `subdir` (puedes usar para ello `stat`, o la combinación de modificadores (`-i` y `-a` de `ls`).
4. Muestra el contenido de `enlaceH` y de `enlaceS` utilizando el comando `cat`. A continuación, borra el fichero `fichero2` y repite el procedimiento. ¿Puedes acceder en ambos casos al contenido del fichero?
5. Utiliza la orden `touch` para modificar las fechas de acceso y modificación del fichero `enlaceH`. ¿Qué cambios se observan en la salida de `stat` tras su ejecución? Investiga a través de la página de manual para modificar únicamente una de dichas fechas (modificación o acceso).

Ejercicio 6: Permisos y modos de apertura

En segundo lugar, trabajaremos con la modificación de permisos de acceso a un determinado fichero. Para ello, consulta la página de manual de la orden `chmod`, y observa los dos modos soportados para especificar los permisos que se desean otorgar a un determinado fichero:

- **Modo simbólico**, que usa un argumento mnemotécnico para especificar los permisos a otorgar a un determinado fichero.

- **Modo octal**, que utiliza un número en base octal para representar el patrón de bits del campo *mode* del i-nodo asociado al fichero.

Experimenta con la orden `chmod` y otorga a cualquier fichero permisos de lectura, escritura o ejecución usando sus dos modos de funcionamiento. Observa las implicaciones del cambio de modo a la hora de leer el contenido del fichero (`cat`) o escribir en él (a través, por ejemplo de `echo`), así como los cambios que se producen en el i-nodo correspondiente (`stat` o `ls -l`).

Por último, es importante diferenciar entre los permisos otorgados a un determinado fichero o directorio (estáticos y almacenados en su i-nodo asociado) y el modo de apertura del mismo desde un programa escrito en C (dinámico y almacenado en las tablas internas del sistema operativo), que restringen, en última instancia, las operaciones que se podrán realizar sobre el fichero desde el proceso en ejecución.

Para ello, escribe un programa en C llamado `apertura.c` (compilado como `apertura.x`) que, utilizando llamadas al sistema POSIX, reúna las siguientes características:

1. El programa recibirá un argumento obligatorio (`-f`) seguido del nombre del fichero a abrir.
2. El programa recibirá, a través de dos argumentos opcionales (`-r` para lectura y `-w` para escritura) el modo de apertura deseado para el fichero. Es necesario proporcionar al menos un modo de apertura, y ambos pueden combinarse para que la apertura sea en modo lectura/escritura.
3. El programa intentará abrir (`open`) el fichero con el modo indicado, reportando error si no es posible. Si el fichero no existe, se creará. En caso contrario, se eliminará todo su contenido.
4. En todo caso, e independientemente del modo de apertura seleccionado, el programa intentará realizar una escritura (`write`) desde el fichero y a continuación una lectura (`read`), reportando un error si no es posible realizar alguna de estas operaciones.
5. Por último, se cerrará el fichero (`close`).

Una vez desarrollado, experimenta al menos con las siguientes situaciones:

1. Otorga permisos de lectura y escritura a un fichero existente en el sistema, y ejecuta el programa `apertura` con las tres opciones disponibles (`-r`, `-w` y `-rw`). ¿Es posible leer y/o escribir en el fichero en todos los casos? ¿Qué función devuelve en este caso el error? ¿Por qué?

2. Elimina el permiso de lectura sobre el fichero destino. ¿Qué función devuelve en este caso el mensaje de error? ¿Por qué? (puedes también hacer lo propio eliminando el permiso de escritura, o los dos simultáneamente).

3. Elimina el permiso de ejecución del fichero ejecutable `apertura.x`. ¿Qué implicación tiene esto de cara a la ejecución del mismo (`./apertura.x`).