Práctica 2: API de ficheros, directorios, procesos e hilos

Índice

1 Objetivos

En esta práctica vamos a hacer varios ejercicos orientados a afianzar nuestro conocimiento del manejo del API POSIX de: ficheros, directorios, procesos e hilos.

La práctica está organizada en 3 partes:

- API de ficheros y directorios
- API de procesos e hilos
- Manejos de ficheros con varios procesos e hilos

Cada parte consiste en uno o más ejercicios independientes. Se aconseja al alumno que cree un directorio por parte con un subdirectorio por ejercicio. En las instrucciones de cada parte se asume que el ejercicio *N* se hace en un subdirectorio llamado *ejercicioN* dentro del directorio común para dicha parte.

El archivo ficheros_p2.tar.gz contiene una serie de ficheros que pueden ser usados como punto de partida para el desarrollo los ejercicios de esta práctica, así como unos makefiles que pueden usarse para la compilación de los distintos proyectos.

2 API de ficheros y directorios

En esta parte trabajaremos las llamadas al sistema: `open`, `read`, `write`, `close`, `stat`, `readlink`, `symlink`, `lseek`, además de algunas funciones de la librería estándar de C que serán necesarias, como `opendir`, `readdir`, `strlen`, `snprintf`, `malloc`y `printf`.

Ejercicio 1: Copia de ficheros regulares

Diseña un programa *copy.c* que permita hacer la copia de un fichero regular. El programa recibira dos parámetros por la línea de comandos. 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. Para ello se reservará un buffer de 512 bytes como almacenamiento intermedio. El programa debe ir leyendo bloques de 512 bytes del fichero origen y escribiendo los bytes leídos en el fichero destino. Debéis tener 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 (es decir, read devolverá menos de 512). Por ello siempre se debe escribir en el fichero destino tantos bytes como se hayan leído del fichero origen (`read` devolverá el número de bytes leídos).

El alumno debe consultar las páginas de manual de las siguientes llamadas al sistema: `open`, `read`, `write` y `close`. En la página de manual de `open` prestad especial atención a los flags de apertura, teniendo que usar en este caso `O_RDONLY`, `O_WRONLY`, `O_CREAT`, `O_TRUNC`.

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 estándar. 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 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 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
```

3 API de procesos e hilos

En esta parte de la práctica trabajaremos las llamadas al sistema: `fork`, `exec`, `wait`, `waitpid`, `getpid`, `getppid`, `sigaction`, `alarm`, `kill`. Además, usaremos las funciones de la biblioteca de pthreads: `pthread_create`, `pthread_join`, `pthread_self`.

Ejercicio 1: Creación de procesos

Diseña un programa *fork1.c* que cree dos procesos hijos. El primero de ellos no cambiará de ejecutable, pero el segundo sí lo hará, mediante una llamada a `execvp`. El programa recibirá como parámetros el nombre del ejecutable que deberá ejecutar y los argumentos que necesite pasarle.

El programa realizará una primera llamada a `fork`. Después de ella, tanto el programa padre como el hijo imprimirán un mensaje indicando si son padre o hijo, su identificador y el de su padre. A continuación, ambos procesos realizarán una segunda llamada a `fork`, después de la cual cada proceso imprimirá un mensaje indicando si es padre o hijo, su identificador y el de su padre. Cada hijo generado en la segunda llamada cambiará su ejecutable por el que se haya pasado como argumento a `main` usando `execvp`. Cada padre esperará a que sus hijos finalicen.

El alumno debe consultar las páginas de manual de las siguientes llamadas al sistema: `fork`, `getpid`, `getppid`, `execvp`, `waitpid`.

Para comprobar el funcionamiento correcto de nuestro programa podemos usar como argumento cualquier ejecutable que imprima algo por pantalla, por ejemplo `echo` o `ls`.

Ejercicio 2: Creación de procesos que cumplan un grafo de dependencias.

En este ejercicio tendremos un proceso padre, que creará *N* hijos siguiendo una grafo de dependencias, de forma que no se pueda crear un determinado hijo hasta que terminen todos los procesos de los que dicho hijo depende. En nuestro caso *N* será 8 y el grafo de dependencias será el del problema 5 de la hoja de procesos. Los hijos se crearán usando

llamadas a fork seguidas de execl (pares) y `execlp` (impares) y cada uno ejecutará el comando , imprimiendo por pantalla su nombre (P0, P1, etc.).

El alumno debe consultar las páginas de manual de las siguientes llamadas al sistema: `fork`, `execl`, `execlp`, `waitpid`.

Ejercicio 3: Creación y paso de parámetros a hilos.

En este ejercicio vamos a usar la biblioteca *pthreads*, por lo que será necesario compilar y enlazar con la opción `-pthread`.

Escribir un programa *hilos.c* que cree un hilo para cada usuario, pasándole a cada uno como argumento el puntero a una estructura que contenga dos campos: un entero, que será el número de usuario, y un caracter, que indicará si el usuario es prioritario (P) o no (N).

El programa deberá crear una variable para el argumento de cada hilo usando memoria dinámica, inicializar dicha variable con el número de usuario y su prioridad (los pares serán prioritarios y los impares no lo serán), crear los hilos y esperar a que finalicen.

Cada hilo copiará sus argumentos en variables locales, liberará la memoria dinámica reservada para los mismos, averiguará cuál es su identificador e imprimirá un mensaje con su identificador, el número de usuario y su prioridad.

El alumno debe consultar las páginas de manual de: `pthread_create`, `pthread_join`, `pthread_self`.

Probar a crear solamente una variable para el argumento de todos los hilos, dándole el valor correspondiente a cada hilo antes de la llamada a `pthread_create`. Explicar qué sucede y cuál es la razón.

Ejercicio 4: Manejo de señales.

En este ejercicio vamos a experimentar el envío de señales, haciendo que un proceso cree a un hijo, espere a una señal de un temporizador y, cuando la reciba, termine con la ejecución del hijo.

Al igual que en el ejercicio 1, el programa principal recibirá como argumento el ejecutable que se desea que ejecute el proceso hijo.

El proceso padre creará un hijo, que cambiará su ejecutable con una llamada a `execvp`. A continuación, el padre éstablecerá que el manejador de la señal `SIGALRM` sea una función que envíe una señal `SIGKILL` al proceso hijo y programará una alarma para que le envíe una señal a los 5 segundos. Antes de finalizar, el padre esperará a que finalice el hijo y comprobará la causa por la que ha finalizado el hijo (finalización normal o por recepción de una señal), imprimiendo un mensaje por pantalla.

El alumno debe consultar las páginas de manual de: `sigaction`, `alarm`, `kill`, `wait`.

Para comprobar el funcionamiento correcto de nuestro programa podemos usar como argumento un ejecutable que termine en menos de 5 segundos (como ls o echo) y uno que no finalice hasta que le llegue la señal (como `xeyes`).

Una vez funcione el programa, modificar el padre para que ignore la señal SIGINT y comprobar que, efectivamente, lo hace.

4 Ejercicio 5: Manejos de ficheros con varios procesos e hilos

Se pretende crear un programa que utilice 10 procesos (el original y 9 procesos hijo) para escribir de manera concurrente un fichero “output.txt”. La idea es que cada proceso escriba una cadena de caracteres con un número decimal repetido 5 veces. Así el proceso inicial escribira 5 ceros (“00000”), el primer proceso hijo 5 unos

("11111"), el segundo 5 doses ("22222") y así sucesivamente. De este modo el contenido del fichero al final será: 0000011111222223333344444555556666677777888899999

Un primer programador con poca experiencia en la programación de sistemas propone la siguiente implementación (fichero *practica_2_5_inicial.c*):

```
int main(void)
{
    int fd1,fd2,i,pos;
    char c;
    char buffer[6];

    fd1 = open("output.txt", O_CREAT | O_TRUNC | O_RDWR, S_IRUSR | S_IWUSR);
    write(fd1, "00000", 5);
    for (i=1; i < 10; i++) {
        pos = lseek(fd1, 0, SEEK_CUR);
        if (fork() == 0) {
            /* Child */
            sprintf(buffer, "%d", i*11111);
            lseek(fd1, pos, SEEK_SET);
            write(fd1, buffer, 5);
            close(fd1);
            exit(0);
        } else {
            /* Parent */
            lseek(fd1, 5, SEEK_CUR);
        }
    }

    //wait for all children to finish
    while (wait(NULL) != -1);

    lseek(fd1, 0, SEEK_SET);
    printf("File contents are:\n");
    while (read(fd1, &c, 1) > 0)
        printf("%c", (char) c);
    printf("\n");
    close(fd1);
    exit(0);
}
```

Tras esta implementación el programador comprueba el funcionamiento, ejecutando 10 veces seguidas el programa con la esperanza de que no se produczcan carreras. El resultado, en la máquina del programador es:

```
$ for i in $(seq 10); do ./practica_2_5_inicial ; done
File contents are:
000001111122222333335555566666888899999
File contents are:
00000111112222255555666668888899999
File contents are:
000001111122222333335555566666888899999
File contents are:
00000111112222224444466666777799999
File contents are:
000004444477777555556666668888899999
File contents are:
000002222244444555557777778888899999
File contents are:
0000011111222224444455555777778888899999
File contents are:
00000111112222255555444448888887777799999
```

```
File contents are:
000001111122222444445555577778888899999
File contents are:
000001111122222555554444488887777799999
```

Al parecer el programa tiene algunos errores, puesto que se producen carreras y el resultado es incorrecto en todos los casos.

Cuestión A: Soluciona la implementación inicial, manteniendo la escritura concurrente en el fichero. Es decir, el proceso padre escribirá los cinco ceros iniciales, el hijo uno los cinco unos, etc, sin necesidad de sincronizar los procesos. Es decir, se desea que no sea necesario imponer un orden en la ejecución de los procesos.

Cuestión B: Proponer una solución en la que el padre escriba su número entre la escritura de los hijos, de modo que el contenido del fichero al final será:

000001111100000222220000033333000004444400000555550000066666000007777700000888880000099999