PRACTICA: Llamadas a Sistema

Objetivos:

-Conocer las llamadas a sistema que gestionan los ficheros

-Crear programas que usen las API de Linux, en vez de las típicas funciones de librería.

Referencias:
http://blog.txipinet.com/index.php/Informatica/
http://www.e-ghost.deusto.es/docs/2006/ProgramacionGNULinux.odt

Lee la siguiente información sobre llamadas a sistema para la gestión de ficheros. Prueba cada uno de los ejemplos. Después de practicar serás capaz de realizar los programas usando dichas APIs.

Llamadas a Sistema para gestionar ficheros

La manera más sencilla de usar llamadas a sistema es usar las más básicas que son las de manejo de ficheros. Ya hemos dicho que en UNIX en general, y en GNU/Linux en particular, todo es un fichero, por lo que estas syscalls son el ABC de la programación de sistemas en UNIX.

Comencemos por crear un fichero. Existen dos maneras de abrir un fichero, open() y creat(). Antiguamente open() sólo podía abrir ficheros que ya estaban creados por lo que era necesario hacer una llamada a creat() para llamar a open() posteriormente. A día de hoy open() es capaz de crear ficheros, ya que se ha añadido un nuevo parámetro en su prototipo:

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

int open( const char *pathname, int flags )

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

Como vemos, la nueva open() es una suma de las funcionalidades de la open() original y de creat(). Otra cosa que puede llamar la atención es el hecho de que el tipo del parámetro "mode" es "mode_t". Esta clase de tipos es bastante utilizado en UNIX y suelen corresponder a un "int" o "unsigned int" en la mayoría de los casos, pero se declaran así por compatibilidad hacia atrás. Por ello, para emplear estas syscalls se suelen incluir los ficheros de cabecera:

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

El funcionamiento de open() es el siguiente: al ser llamada intenta abrir el fichero indicado en la cadena "pathname" con el acceso que indica el parámetro "flags". Estos "flags" indican si queremos abrir el fichero para lectura, para escritura, etc. La siguiente tabla especifica los valores que puede tomar este parámetro:

Indicador	Valor	Descripción
O_RDONLY	0000	El fichero se abre sólo para lectura.
O_WRONLY	0001	El fichero se abre sólo para escritura.

O_RDWR	0002	El fichero se abre para lectura y escritura.
O_SEQUENTIAL	0020	El fichero se abre para ser accedido de forma secuencial (típico de cintas).
O_TEMPORARY	0040	El fichero es de carácter temporal.
O_CREAT	0100	El fichero deberá ser creado si no existía previamente.
O_EXCL	0200	Provoca que la llamada a open falle si se especifica la opción O_CREAT y el fichero ya existía.
O_NOCTTY	0400	Si el fichero es un dispositivo de terminal (TTY), no se convertirá en la terminal de control de proceso (CTTY).
O_TRUNC	1000	Fija el tamaño del fichero a cero bytes.
O_APPEND	2000	El apuntador de escritura se situa al final del fichero, se escribirán al final los nuevos datos.
O_NONBLOCK	4000	La apertura del fichero será no bloqueante. Es equivalente a O_NDELAY.
O_SYNC	10000	Fuerza a que todas las escrituras en el fichero se terminen antes de que se retorne de la llamada al sistema. Es equivalente a O_FSYNC.
O_ASYNC	20000	Las escrituras en el fichero pueden realizarse de manera asíncrona.
O_DIRECT	40000	El acceso a disco se producirá de forma directa.
O_LARGEFILE	100000	Utilizado sólo para ficheros extremadamente grandes.
O_DIRECTORY	200000	El fichero debe ser un directorio.
O_NOFOLLOW	400000	Fuerza a no seguir los enlaces simbólicos. Útil en entornos críticos en cuanto a seguridad.

Tabla 1 - Lista de los posibles valores del argumento “flags”.

La lista es bastante extensa y los valores están pensados para que sea posible concatenar o sumar varios de ellos, es decir, hacer una OR lógica entre los diferentes valores, consiguiendo el efecto que deseamos. Así pues, podemos ver que en realidad una llamada a creat() tiene su equivalente en open(), de esta forma:

```
open( pathname, O_CREAT | O_TRUNC | O_WRONLY, mode )
```

El argumento “mode” se encarga de definir los permisos dentro del Sistema de Ficheros (de la manera de la que lo hacíamos con el comando “chmod”). La lista completa de sus posibles valores es esta:

Indicador	Valor	Descripción
S_IROTH	0000	Activar el bit de lectura para todo los usuarios.
S_IWOTH	0001	Activar el bit de escritura para todo los usuarios.
S_IXOTH	0002	Activar el bit de ejecución para todo los usuarios.
S_IRGRP	0010	Activar el bit de lectura para todo los usuarios pertenecientes al grupo.
S_IRGRP	0020	Activar el bit de escritura para todo los usuarios pertenecientes al grupo.
S_IRGRP	0040	Activar el bit de ejecución para todo los usuarios pertenecientes al grupo.
S_IRUSR	0100	Activar el bit de lectura para el propietario.
S_IWUSR	0200	Activar el bit de escritura para el propietario.
S_IXUSR	0400	Activar el bit de ejecución para el propietario.
S_ISVTX	1000	Activa el “sticky bit” en el fichero.
S_ISGID	2000	Activa el bit de SUID en el fichero.
S_ISUID	4000	Activa el bit de SGID en el fichero.
S_IRWXU	S_IRUSR + S_IWUSR + S_IXUSR	Activar el bit de lectura, escritura y ejecución para el propietario.

S_IRWXG	S_IRGRP + S_IWGRP + S_IXGRP	Activar el bit de lectura, escritura y ejecución para todo los usuarios pertenecientes al grupo.
S_IRWXO	S_IROTH + S_IWOTH + S_IXOTH	Activar el bit de lectura, escritura y ejecución para todo los usuarios.

Tabla 2 - Lista de los posibles valores del argumento “mode”.

Todos estos valores se definen en un fichero de cabecera , por lo que conviene incluirlo:

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

Una llamada correcta a open() devuelve un entero que corresponde al descriptor de fichero para manejar el fichero abierto. Cada proceso maneja una tabla de descriptores de fichero que le permiten manejar dichos ficheros de forma sencilla. Inicialmente las entradas 0, 1 y 2 de esa tabla están ocupadas por los ficheros STDIN, STDOUT y STDERR respectivamente, es decir, la entrada estándar, la salida estándar y la salida de error estándar:

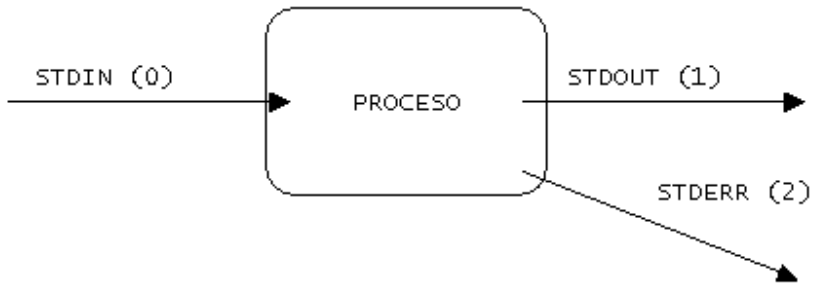


Figura 1 - Los descriptores de fichero iniciales de un proceso.

Podríamos entender esa tabla de descriptores de fichero como un Hotel en el que inicialmente las tres primeras habitaciones están ocupadas por los clientes STDIN, STDOUT y STDERR. Conforme vayan viniendo más clientes (se abran nuevos archivos), se les irá acomodando en las siguientes habitaciones. Así un fichero abierto nada más iniciarse el proceso, es bastante probable que tenga un descriptor de fichero cercano a 2. En este “Hotel” siempre se asigna la “habitación” más baja a cada nuevo cliente. Esto habrá que tomarlo en cuenta en futuros programas.

Bien, ya sabemos abrir ficheros y crearlos si no existieran, pero no podemos ir dejando ficheros abiertos sin cerrarlos convenientemente. Ya sabéis que C se caracteriza por tratar a sus programadores como personas responsables y no presupone ninguna niñera del estilo del recolector de basuras, o similares. Para cerrar un fichero basta con pasarle a la syscall close() el descriptor de fichero como argumento:

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

Ejercicio 1:
Copia, compila y ejecuta el siguiente ejemplo. Debe llamarse ejercicio1.c
La compilación NO debe arrojar warnings. Si los hubiera, incluir las cabeceras necesarias para evitarlo.

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

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

  if( (fd = open( argv[1], O_RDWR )) == -1 )
  {
    perror( "open" );
    exit( -1 );
  }

  printf( "El fichero abierto tiene el descriptor %d.\n", fd );

  close( fd );

  return 0;
}
```

Inicialmente tenemos los ficheros de cabecera necesarios, tal y como hemos venido explicando hasta aquí. Seguidamente declaramos la variable "fd" que contendrá el descriptor de fichero, y realizamos una llamada a open(), guardando en "fd" el resultado de dicha llamada. Si "fd" es –1 significa que se ha producido un error al abrir el fichero, por lo que saldremos advirtiendo del error. En caso contrario se continúa con la ejecución del programa, mostrando el descriptor de fichero por pantalla y cerrando el fichero después. El funcionamiento de este programa puede verse aquí:

```
txipi@neon:~$ gcc ejercicio1.c -o ejercicio1
txipi@neon:~$ ./ejercicio1 ejercicio1.c
El fichero abierto tiene el descriptor 3.
```

El siguiente paso lógico es poder leer y escribir en los ficheros que manejemos. Para ello emplearemos dos syscalls muy similares: read() y write(). Aquí tenemos sus prototipos:

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

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

La primera de ellas intenta leer "count" bytes del descriptor de fichero definido en "fd", para guardarlos en el buffer "buf". Decimos "intenta" porque es posible que en ocasiones no consiga su objetivo. Al terminar, read() devuelve el número de bytes leídos, por lo que comparando este valor con la variable "count" podemos saber si ha conseguido leer tantos bytes como pedíamos o no. Los tipos de datos utilizados para contar los bytes leídos pueden resultar un tanto extraños, pero no son más que enteros en esta versión de GNU/Linux, como se puede ver en el fichero de cabeceras:

```
txipi@neon:~$ grep ssize /usr/include/x86_64-linux-gnu/bits/types.h
typedef int   __ssize_t;    /* Type of a byte count, or error. */
txipi@neon:~$ grep ssize /usr/include/unistd.h
#ifndef __ssize_t_defined
typedef __ssize_t   ssize_t;
# define __ssize_t_defined
[…]
```

El uso de la función write() es muy similar, basta con llenar el buffer "buf" con lo que queramos escribir, definir su tamaño en "count" y especificar el fichero en el que escribiremos con su descriptor de fichero en "fd". Veamos todo esto en acción en un sencillo ejemplo:

Ejercicio 2.
Si no le pasamos parámetros este programa falla. Modifica el programa para que en caso de no encontrar parámetros, le informe al usuario del uso correcto que se espera de este programa.
Nombrarlo como ejercicio2.c

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

#define STDOUT 1
#define SIZE 512

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

        int fd, readbytes;
        char buffer[SIZE];

  if(argv[1] == NULL)  {
      printf("PARA EL CORRECTO USO DEL PROGRAMA HAY QUE PASARLE LA DIRECCION
DE UN ARCHIVO COMO PARAMETRO.\n");
      return -1;
  }

        if( (fd = open( argv[1], O_RDWR )) == -1 )
        {
                perror( "open" );
                exit( -1 );
        }

        while( (readbytes = read( fd, buffer, SIZE )) != 0 )
        {
                write( STDOUT, buffer, readbytes );
        }

        close( fd );

        return 0;
}
```

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

#define STDOUT 1
#define SIZE 512

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

        int fd, readbytes;
        char buffer[SIZE];

        if( (fd = open( argv[1], O_RDWR )) == -1 )
        {
                perror( "open" );
                exit( -1 );
        }

        while( (readbytes = read( fd, buffer, SIZE )) != 0 )
        {
/*              write( STDOUT, buffer, SIZE ); */
                write( STDOUT, buffer, readbytes );
        }

        close( fd );

        return 0;
}
```

Como se puede observar, inicialmente definimos dos constantes, STDOUT para decir que el descriptor de fichero que define la salida estándar es 1, y SIZE, que indica el tamaño del buffer que utilizaremos. Seguidamente declaramos las variables necesarias e intentamos abrir el fichero pasado por parámetro (argv[1]) con acceso de lectura/escritura. Si se produce un error (la salida de open() es –1), salimos indicando el error, si no, seguimos. Después tenemos un bucle en el que se va a leer del fichero abierto ("fd") de SIZE en SIZE bytes hasta que no quede más (read() devuelva 0 bytes leídos). En cada vuelta del bucle se escribirá lo leído por la STDOUT, la salida estándar. Finalmente se cerrará el descriptor de fichero con close().

En resumidas cuentas este programa lo único que hace es mostrar el contenido de un fichero por la salida estándar, parecido a lo que hace el comando "cat" en la mayoría de ocasiones.

Existe una línea de código que está comentada en el listado anterior:

```
/*              write( STDOUT, buffer, SIZE ); */
```

En esta llamada a write() no se está teniendo en cuenta lo que ha devuelto la llamada a read() anterior, sino que se haya leído lo que se haya leído, se intentan escribir SIZE bytes, es decir 512 bytes. ¿Qué sucederá al llamar al programa con esta línea en lugar de con la otra? Bien, si el fichero que pasamos como parámetro es medianamente grande, los primeros ciclos del bucle while funcionarán correctamente, ya que read() devolverá 512 como número de bytes leídos, y write() los escribirá correctamente. Pero en la última iteración del bucle, read() leerá menos de 512 bytes, porque es muy probable que el tamaño del fichero pasado por parámetro no sea múltiplo de 512 bytes. Entonces,

read() habrá leído menos de 512 bytes y write() seguirá tratando de escribir 512 bytes. El resultado es que write() escribirá caracteres “basura” que se encuentran en ese momento en memoria:

Ejercicio 3.
En Ubuntu 20.04 no está tan claro a qué corresponde el tipo ssize_t. Para conocer finalmente a qué equivale este tipo de datos en esta distribución, compila el ejercicio1 con el preprocesador (es decir, convierte el .c en .i) y observa el tipo de datos que devuelve la llamada al sistema write (grep). Indica detalladamente los pasos que has seguido y el tipo de datos equivalente a ssize_t.

```
xabierland@ubuntu:~/Documents/IOS/LAB5$ gcc -o ejercicio2.i -E ejercicio2.c
xabierland@ubuntu:~/Documents/IOS/LAB5$ grep "write" ejercicio2.i
extern ssize_t write (int __fd, const void *__buf, size_t __n) ;
extern ssize_t pwrite (int __fd, const void *__buf, size_t __n,
  char *_IO_write_base;
  char *_IO_write_ptr;
  char *_IO_write_end;
extern size_t fwrite (const void *__restrict __ptr, size_t __size,
extern size_t fwrite_unlocked (const void *__restrict __ptr, size_t __size,
     fd_set *__restrict __writefds,
      fd_set *__restrict __writefds,
  unsigned int __writers;
  unsigned int __writers_futex;
  int __cur_writer;
  write( 1, buffer, readbytes );
xabierland@ubuntu:~/Documents/IOS/LAB5$ grep "ssize_t" ejercicio2.i
typedef long int __ssize_t;
typedef __ssize_t ssize_t;
extern ssize_t read (int __fd, void *__buf, size_t __nbytes) ;
extern ssize_t write (int __fd, const void *__buf, size_t __n) ;
extern ssize_t pread (int __fd, void *__buf, size_t __nbytes,
extern ssize_t pwrite (int __fd, const void *__buf, size_t __n,
extern ssize_t readlink (const char *__restrict __path,
extern ssize_t readlinkat (int __fd, const char *__restrict __path,
extern __ssize_t __getdelim (char **__restrict __lineptr,
extern __ssize_t getdelim (char **__restrict __lineptr,
extern __ssize_t getline (char **__restrict __lineptr,
xabierland@ubuntu:~/Documents/IOS/LAB5$
```

He compilado el programa con gcc -o ejercicio2.i -E ejercicio2.c y mediante grep “write” ejercicio2.i se ve el devuelve un ssize_t y al buscar el typedef de ssize_t de la misma forma vemos que es un long int.

Ejercicio 4.

Guardar el código como ejercicio4.c, compilarlo y ejecutarlo. Cambiar la línea de código que está comentada por la de abajo y volver a compilarlo y ejecutarlo para ver la diferencia.

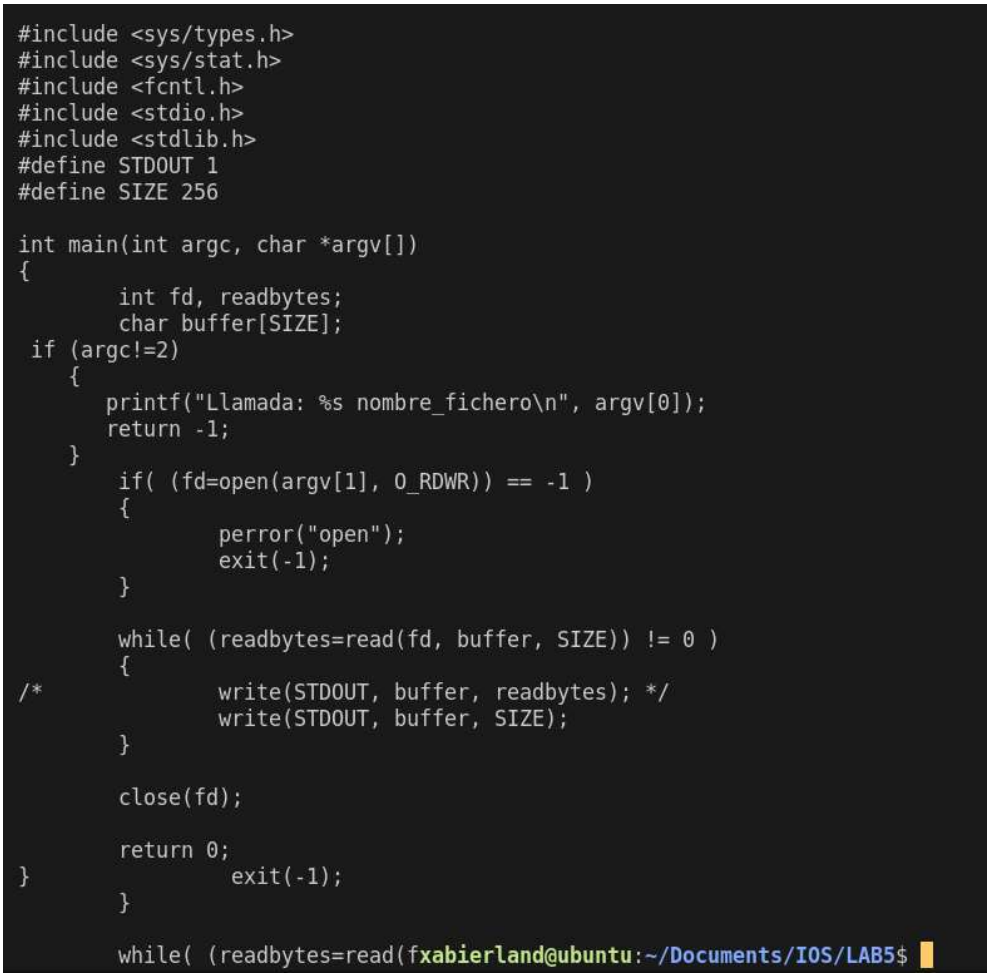
```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#define STDOUT 1
#define SIZE 256

int main(int argc, char *argv[])
{
        int fd, readbytes;
        char buffer[SIZE];
 if (argc!=2)
   {
      printf("Llamada: %s nombre_fichero\n", argv[0]);
      return -1;
   }
        if( (fd=open(argv[1], O_RDWR)) == -1 )
        {
                perror("open");
                exit(-1);
        }

        while( (readbytes=read(fd, buffer, SIZE)) != 0 )
        {
/*              write(STDOUT, buffer, readbytes); */
                write(STDOUT, buffer, SIZE);
        }

        close(fd);

        return 0;
}
```



```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#define STDOUT 1
#define SIZE 256

int main(int argc, char *argv[])
{
        int fd, readbytes;
        char buffer[SIZE];
 if (argc!=2)
    {
       printf("Llamada: %s nombre_fichero\n", argv[0]);
       return -1;
    }
        if( (fd=open(argv[1], O_RDWR)) == -1 )
        {
                perror("open");
                exit(-1);
        }

        while( (readbytes=read(fd, buffer, SIZE)) != 0 )
        {
/*              write(STDOUT, buffer, readbytes); */
                write(STDOUT, buffer, SIZE);
        }

        close(fd);

        return 0;
}               exit(-1);
        }

        while( (readbytes=read(fxabierland@ubuntu:~/Documents/IOS/LAB5$
```

Al usar SIZE en vez de readbytes lo que hacemos es escribir en bloques de 256 bytes. El problema con esto es que el fichero nos muestra caracteres basura como se ve en el ejemplo de arriba. La solucion a esto pasa por usar el readbytes que lo que hace es leer el numero exactos de bytes leidos por la funcion read y escribir ese mismo numero. Por lo tanto si el ultimo bloque es inferior a los 256 solo escribira esa parte de bytes.

```
txipi@neon:~ $ gcc files.c -o files
txipi@neon:~ $ ./files files.c
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#define STDOUT 1
#define SIZE 256

int main(int argc, char *argv[])
{
       int fd, readbytes;
       char buffer[SIZE];
 if (argc!=2)
   {
      printf("Llamada: %s nombre_fichero\n", argv[0]);
      return -1;
   }
       if( (fd=open(argv[1], O_RDWR)) == -1 )
       {
              perror("open");
              exit(-1);
       }

       while( (readbytes=read(fd, buffer, SIZE)) != 0 )
       {
/*             write(STDOUT, buffer, readbytes); */
              write(STDOUT, buffer, SIZE);
       }

       close(fd);

       return 0;
}
```

Tal y como muestra este ejemplo, inicialmente el programa funciona bien, pero si no tenemos en cuenta los bytes leídos por read(), al final terminaremos escribiendo caracteres “basura”.

Otra función que puede ser de gran ayuda es lseek(). Muchas veces no queremos posicionarnos al principio de un fichero para leer o para escribir, sino que lo que nos interesa es posicionarnos en un desplazamiento concreto relativo al comienzo del fichero, o al final del fichero, etc. La función lseek() nos proporciona esa posibilidad, y tiene el siguiente prototipo:

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

Ejercicio 5

lseek devuelve off_t. ¿Cómo puedes saber de qué tipo es off_t?

```
xabierland@ubuntu:~/Documents/IOS/LAB5$ gcc -o ejercicio6.i -E ejercicio6.c
xabierland@ubuntu:~/Documents/IOS/LAB5$ grep "off_t" ejercicio6.i
typedef long int __off_t;
typedef __off64_t __loff_t;
typedef __off_t off_t;
extern __off_t lseek (int __fd, __off_t __offset, int __whence) __attribute__ ((__nothrow__ , __leaf__));
        __off_t __offset) ;
         __off_t __offset) ;
extern int truncate (const char *__file, __off_t __length)
extern int ftruncate (int __fd, __off_t __length) __attribute__ ((__nothrow__ , __leaf__)) ;
extern int lockf (int __fd, int __cmd, __off_t __len) ;
typedef __loff_t loff_t;
    __off_t st_size;
    __off_t l_start;
    __off_t l_len;
extern int posix_fadvise (int __fd, off_t __offset, off_t __len,
extern int posix_fallocate (int __fd, off_t __offset, off_t __len);
  __off_t __pos;
  __off_t _old_offset;
extern int fseeko (FILE *__stream, __off_t __off, int __whence);
extern __off_t ftello (FILE *__stream) ;
xabierland@ubuntu:~/Documents/IOS/LAB5$
```

De la misma forma que hemos buscando el ssize_t podemos pasar el el ejercicio6.c a ejercicio6.i y usando grep buscar por la typedef del tipo de dato off_t que como podemos ver vuelve a ser un long int.

Los parámetros que recibe lseek son bien conocidos, "fildes" es el descriptor de fichero, "offset" es el desplazamiento en el que queremos posicionarnos, relativo a lo que indique "whence", que puede tomar los siguientes valores:

Indicador	Valor	Descripción
SEEK_SET	0	Posiciona el puntero a "offset" bytes desde el comienzo del fichero.
SEEK_CUR	1	Posiciona el puntero a "offset" bytes desde la posición actual del puntero..
SEEK_END	2	Posiciona el puntero a "offset" bytes desde el final del fichero.

Tabla 3 - Lista de los posibles valores del argumento "whence".

Por ejemplo, si queremos leer un fichero y saltarnos una cabecera de 200 bytes, podríamos hacerlo así:

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

#define STDOUT 1
#define SIZE 512

int main( int argc, char *argv[] )
{
   int fd, readbytes;
   char buffer[SIZE];

   if( (fd = open( argv[1], O_RDWR )) == -1 )
   {
      perror( "open" );
      exit( -1 );
   }

   lseek( fd,200, SEEK_SET );

   while( (readbytes = read( fd, buffer, SIZE )) != 0 )
   {
      write( STDOUT, buffer, SIZE );
   }

   close(fd);

   return 0;
}
```

Ejercicio 6

Modifica el código anterior para que no muestre basura al final del write.

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

#define STDOUT 1
#define SIZE 512

int main( int argc, char *argv[] )
{
   int fd, readbytes;
   char buffer[SIZE];

   if( (fd = open( argv[1], O_RDWR )) == -1 )
   {
      perror( "open" );
      exit( -1 );
   }

   lseek(fd, 11, SEEK_SET );

   while( (readbytes = read( fd, buffer, SIZE )) != 0 )
   {
      write( STDOUT, buffer, SIZE );
   }

   close(fd);

   return 0;
}
```

Como el ls -l ejercicio6.c nos indica que tiene 523 bytes y nosotros buscamos en paquetes de 512 bytes nos colocamos 11 posiciones por delante del principio para que imprima exactamente hasta el ultimo carácter aunque esto provoca que el primer include no salga completo.

Ya sabemos crear, abrir, cerrar, leer y escribir, ¡con esto se puede hacer de todo! Para terminar con las funciones relacionadas con el manejo de ficheros veremos chmod(), chown() y stat(), para modificar el modo y el propietario del fichero, o acceder a sus características, respectivamente.

La función chmod() tiene el mismo uso que el comando del mismo nombre: cambiar los modos de acceso permitidos para un fichero en concreto. Por mucho que estemos utilizando C, nuestro programa sigue sujeto a las restricciones del Sistema de Ficheros, y sólo su propietario o root podrán cambiar los modos de acceso a un fichero determinado. Al crear un fichero, bien con creat() o bien con open(), éste tiene un modo que estará en función de la máscara de modos que esté configurada (ver "man umask"), pero podremos cambiar sus modos inmediatamente haciendo uso de una de estas funciones:

```
int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

Viendo el prototipo de cada función, podemos averiguar su funcionamiento: la primera de ellas, chmod(), modifica el modo del fichero indicado en la cadena "path". La segunda, fchmod(), recibe un descriptor de fichero, "fildes", en lugar de la cadena de caracteres con la ruta al fichero. El parámetro "mode" es de tipo "mode_t", pero en GNU/Linux es equivalente a usar una variable de tipo entero. Su valor es exactamente el mismo que el que usaríamos al llamar al comando "chmod", por ejemplo:

```
chmod( "/home/txipi/prueba", 0666 );
```

Para modificar el propietario del fichero usaremos las siguientes funciones:

```
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

Con ellas podremos cambiar el propietario y el grupo de un fichero en función de su ruta (chown() y lchown()) y en función del descriptor de fichero (fchown()). El propietario ("owner") y el grupo ("group") son enteros que identifican a los usuarios y grupos, tal y como especifican los ficheros "/etc/passwd" y "/etc/group". Si fijamos alguno de esos dos parámetros ("owner" o "group") con el valor –1, se entenderá que deseamos que permanezca como estaba. La función lchown() es idéntica a chown() salvo en el tratamiento de enlaces simbólicos a ficheros. En versiones de Linux anteriores a 2.1.81 (y distintas de 2.1.46), chown() no seguía enlaces simbólicos. Fue a partir de Linux 2.1.81 cuando chown() comenzó a seguir enlaces simbólicos y se creó una nueva syscall, lchown(), que no seguía enlaces simbólicos. Por lo tanto, si queremos aumentar la seguridad de nuestros programas, emplearemos lchown(), para evitar malentendidos con enlaces simbólicos confusos.

Ejercicio 7
Explica con un ejemplo en qué consiste el bit SUID y SGID.
Ayuda: http://es.wikipedia.org/wiki/Setuid
SUID y SGID son dos bytes de ficheros y directorios que permiten a diferentes usuarios y grupos trabajar en otro ficheros con fasicilidad.
El SUID (4XXX) es el encargado de que cualquier usuario que ejecute un archivo tenga los permisos necesarios para ello. Esto lo hace convirtiendo al usuario que lo ejecuta en el usuario que ha creado el archivo para asi tenga todos los permisos sobre el. Por ejemplo si trabajamos con el /usr/bin/passwd veremos que tiene permisos **4**701 por lo que permite a todo usuario cambiar su contraseña modificando tambien otros ficheros como el shadow que solo el root podria modificar. El SUID en ficheros no hace nada.

El SGID (2XXX) en ficheros funciona igual que es SUID. En directorios, en cambio, hace que las subcarpetas o ficheros que se creen dentro una carpeta con este bit sean del mismo grupo y asi hace que no sean intratables por el usuario.

Cuando el propietario de un fichero ejecutable es modificado por un usuario normal (no root), los bits de SUID y SGID se deshabilitan. El estándar POSIX no especifica claramente si esto debería ocurrir también cuando root realiza la misma acción, y el comportamiento de Linux depende de la versión del kernel que se esté empleando. Un ejemplo de su uso podría ser el siguiente:

```
gid_t grupo = 100; /* 100 es el GID del grupo users */

chown( "/home/txipi/prueba", -1, grupo);
```

Con esta llamada estamos indicando que queremos modificar el propietario y grupo del fichero "/home/txipi/prueba", dejando el propietario como estaba (-1), y modificando el grupo con el valor 100, que corresponde al grupo "users":

```
txipi@neon:~$ grep 100 /etc/group
users:x:100:
```

Ya sólo nos queda saber cómo acceder a las características de un fichero, mediante el uso de la función stat(). Esta función tiene un comportamiento algo diferente a lo visto hasta ahora: utiliza una estructura de datos con todas las características posibles de un fichero, y cuando se llama a stat() se pasa una referencia a una estructura de este tipo. Al final de la syscall, tendremos en esa estructura todas las características del fichero debidamente cumplimentadas. Las funciones relacionadas con esto son las siguientes:

```
int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
```

Es decir, muy similares a chown(), fchown() y lchown(), pero en lugar de precisar los propietarios del fichero, necesitan como segundo parámetro un puntero a una estructura de tipo "stat":

```
struct stat {
  dev_t     st_dev;     /* dispositivo */
  ino_t     st_ino;     /* numero de inode */
  mode_t    st_mode;    /* modo del fichero */
  nlink_t   st_nlink;   /* numero de hard links */
```

```
  uid_t     st_uid;   /* UID del propietario*/
  gid_t     st_gid;   /* GID del propietario */
  dev_t     st_rdev;  /* tipo del dispositivo */
  off_t     st_size;  /* tamaño total en bytes */
  blksize_t st_blksize; /* tamaño de bloque preferido */
  blkcnt_t  st_blocks;  /* numero de bloques asignados */
  time_t    st_atime;   /* ultima hora de acceso */
  time_t    st_mtime;   /* ultima hora de modificación */
  time_t    st_ctime; /* ultima hora de cambio en inodo */
};
```

Ejercicio 8

Cada fichero (y por tanto cada directorio, en Unix se tratan de forma similar) tiene asociado una entrada en el sistema de ficheros conocida como inode (o inodo).

Con la opción -i del comando ls es posible conocer ese número.

Crea 2 ficheros de nombre p1 y p2 en el directorio /tmp y obtén sus correspondientes inodos.

```
xabierland@ubuntu:~$ touch /var/tmp/p1
xabierland@ubuntu:~$ touch /var/tmp/p2
xabierland@ubuntu:~$ ls -i /var/tmp/
526094 p1
526320 p2

xabierland@ubuntu:~/Documents/IOS/LAB5$ ./ejercicio8 /var/tmp/p1
Propiedades del fichero </var/tmp/p1>
i-nodo: 526094
modo: 0100664
vinculos: 1
tipo del dispositivo: 0
tamaño total en bytes: 0
tamaño de bloque preferido: 4096
numero de bloques asignados: 0

xabierland@ubuntu:~/Documents/IOS/LAB5$ ./ejercicio8 /var/tmp/p2
Propiedades del fichero </var/tmp/p2>
i-nodo: 526320
modo: 0100664
vinculos: 1
tipo del dispositivo: 0
tamaño total en bytes: 0
tamaño de bloque preferido: 4096
numero de bloques asignados: 0
xabierland@ubuntu:~/Documents/IOS/LAB5$
```

Como vemos, tenemos acceso a información muy detallada y precisa del fichero en cuestión. El siguiente ejemplo muestra todo esto en funcionamiento:

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

```
int main( int argc, char *argv[] )
{
  struct stat estructura;

  if( ( lstat( argv[1], &estructura ) ) < 0 )
  {
    perror( "lstat" );
    exit( -1 );
  }
  printf( "Propiedades del fichero <%s>\n", argv[1] );
  printf( "i-nodo: %ld\n", estructura.st_ino );
  printf( "modo: %#o\n", estructura.st_mode );
  printf( "vinculos: %ld\n", estructura.st_nlink );
  printf( "tipo del dispositivo: %ld\n", estructura.st_rdev );
  printf( "tamaño total en bytes: %ld\n", estructura.st_size );
  printf( "tamaño de bloque preferido: %d\n",          estructura.st_blksize );
  printf( "numero de bloques asignados: %ld\n",          estructura.st_blocks );

  return 0;
}
```

Hay algunos detalles destacables en el anterior código:

- Utilizamos “%#o” para mostrar de forma octal el modo de acceso del fichero, sin embargo aparecen más cifras octales que las 4 que conocemos. Esto es porque también se nos informa en ese campo del tipo de fichero (si es un directorio, un dispositivo de bloques, secuencial, un FIFO, etc.).