Sistemes Operatius II

Informe de la Practica 1

Realizado por Joaquim Yuste y Marcos Plaza

Índice

Introducción

En esta practica nos centraremos en recordar algunas de las operaciones que utilizan *llamadas a sistema* (*open, close, write, read* i *lseek*), así como las operaciones de la librería '*stdio.h*' (*fopen, fclose, fwrite, fread, fprintf, fscanf* i *fseek*), y las operaciones para mapear un archivo en memoria las cuales también hacen uso de las *llamadas a sistema* (*mmap, munmap* o *stat*, entre otras).

Hace falta decir que hemos dividido el informe en cuatro secciones, las cuales cada una corresponde a uno de los apartados del enunciado de la práctica. En cada apartado vamos a explicar qué funciones proporcionadas en el '*Campus Virtual*' de la asignatura vamos a usar.

Es importante refrescar estos conceptos para la correcta realización de las futuras prácticas, así que al inicio de cada sección nos ha parecido una buena idea analizar a fondo el funcionamiento de cada método.

Informe de la práctica

Primer apartado. Uso de las operaciones que realizan *llamadas a sistema*

Inicialmente revisaremos las funciones que realizan *llamadas a sistema* (solicitamos el permiso para realizar alguna instrucción al *sistema operativo*), destacando el uso de *open, write, read* y *close* mediante los programas proporcionados; *write_int.c, read_int.c y write_char_int.c*.

Pero primero veamos como funcionan las operaciones anteriormente mencionadas:

1) open (char *path, int offlags)
   Llamada a sistema que abre un nuevo archivo y te devuelve su descriptor de fichero para poder trabajar con él. El valor del offlags determinará el método de abertura del fichero (solo lectura, solo escritura...).

2) close (int fd)
   Llamada a sistema que cierra un archivo abierto con anterioridad haciendo uso de su descriptor de fichero.

3) read (int fd, void *buf, size_t nBytes)
   Llamada a sistema que te permite guardar en un buffer tantos Bytes de información como le especifiques, estos Bytes los leera del archivo al que haga referencia del descriptor de fichero.

4) write (int fd, void *buf, size_t nBytes)
   Llamada a sistema que escribe a un archivo una cantidad especificada de Bytes, la información a escribir será la contenida en el buffer.

A continuación nos dispondremos a ejecutar los códigos *write_int.c*, *read_int.c* y *write_char_int.c* tal y como nos piden.

En esta primera cuestión nos preguntan sobre cuál será el tamaño del archivo generado por *write_int.c* para un determinado valor de N. Teniendo en cuenta que este código escribe en un archivo tantos enteros (cada entero ocupa 4 Bytes) como N introduzcas, el tamaño del archivo será de la forma: N * 4 Bytes.
Al ejecutar el programa, comprobamos que el tamaño es el esperado, además, también podemos ver que sin cerrar el archivo este ya ocupa lo que debería, esto se debe a que la llamada a sistema write escribe directamente sobre disco. Por último, una vez cerramos él archivo su tamaño no varía.

Ahora intentamos leer este mismo archivo con el código *read_int.c*. Le pasamos el nombre del fichero por parámetro, al ejecutarlo nos imprime por pantalla tantos números como N habíamos puesto en el programa anterior, es decir, *read_int.c* ha sido capaz de leer correctamente la información que habíamos escrito anteriormente ya que contiene un bucle for que va recogiendo de 4 en 4 Bytes (de entero en entero).

El siguiente código que nos proporcionan es el *write_char_int.c*, al analizarlo podemos ver que escribe en un archivo una cadena de caracteres acompañada de un entero, las N veces que especifiquemos. La cadena que escribe corresponde a “so2” (un byte por cada carácter) y, por lo tanto, estaremos escribiendo 3 Bytes * N + 4 Bytes * N, que en general lo podemos expresar cómo: 7 Bytes * N. Al igual que el *write_int.c* nada más ejecutarlo el archivo ya tiene el tamaño esperado y al cerrarlo este no varía.

Si abrimos el archivo generado con el programa *okteta* podemos observar que aparecen muchos números en hexadecimal. Observamos también otra ventana con los caracteres correspondientes según el código ASCII (o otros sistemas de codificación como ISO-8859-1 o UTF-8). Podemos ver que los caracteres 's', 'o' y '2' corresponden a 73, 6F y 32 respectivamente en hexadecimal según el código ASCII. A continuación viene la

representación en ASCII de los enteros que nos encontramos después de cada so2. Podemos observar que el byte de menos peso se encuentra en la posición del byte de más peso. Esto es así ya que algunos procesadores trabajan con datos de mas de un byte ordenados de una forma u otra; esto se denota con el nombre de Endianess. Podemos ver en este caso que nuestro procesador Intel invierte el orden natural de los Bytes y para cada entero pone el Byte de menos peso en primer lugar (little endian). Podemos ver que la representación de estos números en hexadecimal según el código ASCII corresponden a unos caracteres extraños en lugar de su representación como número.

Para finalizar este apartado, intentamos leer el archivo generado con el programa *write_char_int.c* con el código *read_int.c*. Al hacerlo nos damos cuenta que nos imprime por pantalla números sin sentido (a simple vista). El propósito de *read_int.c* es leer los Bytes de cuatro en cuatro e interpretarlos como un entero, por lo tanto vamos a obtener en el primer caso que va a coger (observando *okteta*) los 4 primeros Bytes (73 6F 32 00), que corresponde exactamente al primer número que se imprime en terminal al ejecutar el programa 3305331.

Segundo apartado. Uso de algunas de las funciones de la librería de usuario estándar '*stdio.h*'

Tal y como hemos hecho anteriormente primero nos pararemos a analizar el funcionamiento de las funciones que utilizaremos de esta librería:

1) fopen (char *path, char *mode)
   Función que abre el archivo pasado por parámetro utilizando el modo especificado (lectura, escritura, ambos...), al hacerlo devuelve un puntero de fichero de la clase FILE, en caso de no haber podido abrirlo devolverá NULL.

2) fwrite (void *ptr, size_t size, size_t nmemb, FILE *stream)
   Función que escribe información del puntero introducido a un buffer interno del al puntero del objeto FILE correspondiente, una vez el buffer se llena transmite la información a disco para minimizar las llamadas a sistema.
   size: número de Bytes de cada elemento.
   nmemb: número de elementos a escribir.

3) fread (void *ptr, size_t size, size_t nmemb, FILE *stream)
   Función que guarda información del del objeto FILE a su buffer interno y la va enviando según la necesite al puntero de destino para minimizar las llamadas a sistema.
   size: número de Bytes de cada elemento.
   nmemb: número de elementos a escribir.

4) fclose(FILE *stream)
   Cierra el archivo y, si es necesario, escribe la información del buffer interno a disco.

Una vez analizadas las funciones que necesitaremos ejecutamos el código *fwrite_int.c* con los valores propuestos en el enunciado.

Para los tres primeros valores (10, 100, 1000) el tamaño del fichero antes de cerrarlo es de 0 Bytes. Sin embargo, para N = 2000 el tamaño del fichero antes de cerrarlo es de 4kB, esto se debe a que el objeto del tipo FILE contiene un buffer interno que va guardando los datos y una vez se llena hace la llamada a sistema write(). Aun así, para todos los valores, una vez cerramos el fichero, el tamaño del archivo es el que debería (N * 4 Bytes) y por lo tanto podremos saber el tamaño real del archivo antes de cerrarlo.

En el momento de leer los enteros escritos con el código *fwrite_int.c* (en este caso nos piden que sean 100 enteros), podemos hacer uso tanto de los códigos *read_int.c* como *fread_int.c*. pero no los dos serán igual de eficientes.

El código *read_int.c* hará 100 llamadas a sistema (lo cual es bastante costoso), sin embargo *fread_int.c* solo generará una, guardará los 100 *4 Bytes en el buffer de una sola vez e irá recogiendo la información de este según se necesite.

Tercer apartado. Uso de algunas de las funciones de la librería de usuario estándar '*stdio.h*'

Nos disponemos explicar las nuevas funciones que se necesitarán para los siguientes ejercicios:

1) fprintf (FILE *stream, const char *format, ...)
   Escribe texto en un fichero siguiendo un cierto código de formato. Si no se indican suficientes argumentos para completar los códigos de formato, el resultado es indeterminado. Devuelve el número de caracteres escritos.
   stream: es el puntero al objeto FILE que usaremos para utilizar las funciones de la librería de usuario estándar.
   format: este puntero contiene la sequencia de caracteres que se va a escribir en el archivo. Se utilizan otros caracteres para dar un formato concreto a lo que se quiere escribir para a continuación pasarlos por parámetro.

2) fscanf (FILE *stream, const char *format, ...)
   Lee valores desde teclado, siguiendo un cierto código de formato.
   stream: es el puntero al objeto FILE que usaremos para utilizar las funciones de la librería de usuario estándar.
   format: este puntero contiene la sequencia de caracteres que vamos a recoger del archivo. Especificamos en que formato queremos recoger los datos del archivo.

La intención de este tercer apartado es la de ver las diferencias entre las diferentes funciones de la librería estándar '*stdio*'. Haremos uso de las funciones *fwrite_int.c, fprintf_int.c* y *fscanf_int.c*.

En primer lugar se nos pide que ejecutemos la función *fwrite_int.c* para una N = 100, la cual usa la función *fwrite* de la '*stdio*' para escribir en el archivo. Dicha función nos ha escrito en este archivo 100 números enteros de 4 Bytes cada uno. En abrir este fichero en un editor de texto (en nuestro caso, el editor de texto que viene por defecto en Ubuntu) no se nos muestran los números normales del 0 al 99 sino que aparecen otros caracteres extraños. Los caracteres que nos va a mostrar un editor de texto cualquiera van a depender, en cierto modo de cómo hayamos configurado nuestro teclado o la codificación que use el editor de texto en particular. En nuestro caso el editor de texto que se abre por defecto usa UTF-8 que viene a ser en cierto modo casi lo mismo que la codificación US-ASCII. Hasta que no aparece el numero 21 no se empieza a mostrar ningún carácter ya que el primer carácter visible en la tabla del código ASCII es el signo de exclamación '!' (Hex: 21).

Ahora usaremos la función *fprintf_int.c* para el mismo número de N. En este caso internamente no se han escrito enteros, sino que los datos han sido formateados y se han escrito directamente los caracteres en US-ASCII. Podemos observar en el *okteta* que el primer dígito en hexadecimal (30) corresponde al 0 en US-ASCII (cada carácter va acompañado por el numero A0 que corresponde al salto de línea) por lo tanto aquí se puede apreciar la diferencia con la función *fwrite_int.c*. Tenemos que internamente el *fprintf* de la librería estándar de usuario '*stdio*', esta formateando los enteros que le vamos pasando y los convierte a un carácter con lo cual podemos observar la representación correcta de cada número en decimal.

Si hiciéramos que *read_int.c* leyese los caracteres escritos en el archivo generado por la función *fprintf_int.c* para N = 100 veríamos que en el terminal se nos van a imprimir números bastante grandes y a priori sin sentido (ya que queremos que los números que se impriman en terminal sean los del intervalo del 0 al 99). Eso es así por qué debemos recordar que el *read_int* va a ir recogiendo los datos de 4 en 4 Bytes sin hacer el formateo inverso de US-ASCII a entero para a continuación imprimir por pantalla.

La ultima cuestión que se nos plantea en este apartado es la de sí podemos utilizar el programa de *fscanf_int.c* para leer los datos generados por el *fwrite_int.c*. Tenemos que el archivo generado por el *fwrite_int.c* es un archivo que se almacena en disco tal y como esta en memoria, mientras que *fprintf* genera un archivo que en disco se guarda formateado para que cada entero escrito pueda leerse como un símbolo del código ASCII. Dentro de la función *fscanf_int.c,* cuando se usa el *fscanf* podremos observar que le damos por parámetro el formato al que queremos reformatear la información que tenemos en el archivo (es compatible pasar los caracteres ascii a char o a entero por ejemplo), pero sin embargo no podemos reformatear los números enteros "puros" (ya que solo es compatible de ASCII a otros tipos de dato) y devolverlos a entero, para ello usaremos la función *read_int.c* como hemos visto anteriormente.

Cuarto apartado. Uso de las funciones para mapear ficheros en memoria

Por último en esta sección vamos a tratar con funciones que también realizan llamadas a sistema como son la mmap o munmap, para mapear ficheros a memoria y así escribir en ellos. Veremos qué se emplea un método diferente para escribir y leer en este caso. Vamos a echar un vistazo a las últimas operaciones que nos gustaría repasar:

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

   La función mmap intenta ubicar length bytes comenzando en el desplazamiento offset desde el fichero (u otro objeto) especificado por el descriptor de fichero fd en memoria preferiblemente en la dirección start. Esta última dirección es una sugerencia y normalmente se especifica como 0. El lugar donde es ubicado el objeto es devuelto por mmap, y nunca vale 0. El argumento prot describe la protección de memoria deseada. (y no debe entrar en conflicto con el modo de apertura del fichero).

2) munmap (void *start, size_t length)

   La llamada al sistema munmap borra las ubicaciones para el rango de direcciones especificado, y produce referencias a las direcciones dentro del rango a fin de generar referencias a memoria inválidas. La región es también desubicada automáticamente cuando el proceso termina.

En este experimento usaremos el archivo generado por el *fwrite_int.c* para a continuación leerlo con el *mmap_read_int.c*. Podemos observar que la lectura del archivo se realiza de forma correcta ya que como ya sabemos el *fwrite_int.c* ha escrito 100 valores en el archivo (en disco) tal y como previamente se almacenan en memoria, por lo tanto al traer de vuelta estos valores y mostrarlos (con printf) no representa ningún problema.

Ahora vamos a usar un archivo generado por el *mmap_write_int.c* (para N = 100) y lo vamos a intentar leer con el *fread_int.c* y el *fscanf_int.c*. Podemos observar que en la función *mmap_write_int.c* se "reserva" el espacio que va a ser usado por el fichero y en este se escribe todo tal cual se escribe en memoria (un entero corresponde a 4 bytes). Por lo tanto al leer con el *read_int.c* (que recordemos qué va cogiendo de 4 en 4 bytes) no habrá ningún problema para después mostrarlos. Sin embargo sí usamos el *fscanf_int.c* nos ocurrirá como en el caso anterior cuando queríamos leer de la función *fwritef_int.c*. Al ser *fscanf* una función que traslada el formato ASCII a el tipo de dato que le corresponde en cada caso (según lo que le pasemos por parámetro) no va a poder realizarse la conversión de los números del archivo y como consecuencia no va a poder mostrarlos.

Para finalizar vamos a echar el ojo a el código del *mmap_write_int.c*, más particularmente a las siguientes lineas de código:

```
/* Codi preguntat a la practica */

        lseek(fd, len-1, SEEK_SET); //Colocamos el puntero al final del archivo
        write(fd, "", 1); //Escribimos un carácter en blanco - da igual lo que escribas mientras se escriba un
      solo byte

/* Fi codi */
```

Lo que realiza esta parte del código es una especie de "reserva de espacio" (podríamos verlo de forma similar a lo que hace la función malloc) para lo que va a ser el archivo que vamos a mapear en memoria, es decir vamos a generar el archivo inicialmente para después poder mapearlo en memoria y escribir en el. Con lseek vamos a realizar esta "reserva de espacio o resize" para el fichero situando el puntero a casi el final de este, justo antes de el ultimo byte que necesitaremos para escribir los enteros. A continuación con él write escribiremos un carácter vacío, es decir, escribiremos el "\0" para finalizar con la medida máxima del fichero. Si quitamos esta parte del código no escribiremos nada en el archivo por que su dimesión será 0 y por lo tanto al mapearlo en memoria la dimensión del vector también será 0, con lo cual cuando intentemos escribir en este último obtenemos *error en el bus (core generado)*, algo parecido al la típica violación de segmento.

Conclusiones tras realizar la practica

A través de la realización de esta práctica hemos sido capaces de ponernos al día con las funciones típicas para la manipulación de ficheros y que usaremos mas adelante en los siguientes trabajos.