Universidad de Alcalá

Departamento de Automática

Sistemas Operativos

Máster Universitario en Ingeniería de Telecomunicación

Práctica 2
Llamadas al sistema

Sebastián Sánchez Prieto

1 Objetivos

- Usar las llamadas al sistema básicas de manejo de E/S y sistema de archivos (`open`, `close`, `read`, `write`, `lseek`, `fstat`), llamadas relacionadas con la medida del tiempo (`gettimeofday`) y llamadas relacionadas con la proyección de archivos en memoria (`mmap` y `munmap`).
- Estudiar el rendimiento de dos métodos de acceso a archivos (E/S clásica y E/S mediante proyección de archivos en memoria) mediante diversas pruebas.

2 Descripción de la práctica

2.1 Introducción

Esta práctica va a servir por un lado para afianzar los conocimientos del alumno sobre el modo de desarrollar un programa en un sistema UNIX/Linux y las herramientas que se utilizan, y por otro para realizar un análisis de cómo influye en el rendimiento la forma en que se realiza el acceso a archivos en un programa, introduciendo una técnica avanzada como es la proyección de archivos en memoria. Para ello, el alumno desarrollará un mismo programa empleando diferentes llamadas al sistema y comprobará los tiempos de ejecución en cada caso, dando una explicación razonada de los resultados.

Durante la práctica se utilizarán varias llamadas al sistema que cubren diversos servicios del sistema operativo. A continuación se relacionan las llamadas que será necesario emplear para el desarrollo de la práctica dando una breve introducción a cada una de ellas.

Debe tenerse en cuenta que la información presentada está resumida; para ampliar información y conocer el funcionamiento exacto de cada llamada, el alumno debe referirse siempre a las páginas del manual (**`man`**), en particular a la sección 2 del mismo.

2.2 Llamadas al sistema para manejo de archivos

2.2.1 Introducción

En UNIX, todas las operaciones sobre un archivo se realizan empleando el denominado *descriptor de archivo*, mantenido por el sistema operativo. Cuando queremos utilizar un archivo debemos indicarlo mediante una llamada al sistema **`open`** y el sistema operativo creará el descriptor de archivo necesario, devolviendo al usuario un número de descriptor (de tipo entero) que se empleará en lo sucesivo

para realizar operaciones sobre el archivo.

2.2.2 open

La declaración de **open** es:

```
int open (const char *camino, int flags, mode_t modo);
```

donde:

camino	Nombre del archivo.
flags	Modo de apertura; para lectura, escritura, etc.
mode	Permisos del archivo en caso de que éste se deba crear.

La llamada devuelve un entero que correspondiente al número de descriptor del archivo, o bien -1 en caso de error.

2.2.3 read

Esta llamada se emplea para leer datos de un archivo y guardarlos en un *buffer* en memoria. Su declaración es:

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

donde:

fd	Descriptor de archivo sobre el que realizar la operación.
buf	Dirección del *buffer* donde se van a colocar los datos leídos.
nbytes	Número de bytes que el usuario quiere leer.

La llamada devuelve un entero con el número de bytes leídos si es positivo, o -1 en caso de error. Obsérvese, que si el archivo tiene menos bytes disponibles que los solicitados, este número será diferente de **nbytes**.

2.2.4 write

Esta llamada se emplea para escribir en un archivo los datos que tenemos almacenados en un *buffer* de memoria. Su sintaxis es la siguiente:

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

donde el significado de las variables es el mismo que para la llamada al sistema `read`. En cuanto al valor retornado, son los bytes escritos (0 si no se ha escrito ningun dato), o -1 en caso de error.

2.2.5 lseek

Esta llamada se emplea para posicionar el puntero de lectura-escritura asociado a un archivo. Su sintaxis es:

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

donde:

`fildes` Descriptor de archivo sobre el que realizar la operación.

`offset` Desplazamiento del puntero de lectura-escritura.

`whence` Posición de referencia del desplazamiento.

El valor devuelto es igual a la posición donde se ubica el puntero de lectura-escritura. Se recomienda consultar la página del manual para información adicional acerca de los posibles valores de **`whence`** y su significado.

2.2.6 close

Un proceso cierra un archivo cuando no lo va a utilizar más. La sintaxis de la llamada es:

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

donde `fd` es el descriptor del archivo y se devuelve 0 en caso de éxito o -1 en caso de error.

2.2.7 stat, fstat y lstat

Estas llamadas permiten obtener gran cantidad de información sobre un archivo, como por ejemplo su tamaño, propietario, fechas de creación y acceso, etc. Su sintaxis es:

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

donde:

`path` Ruta del archivo sobre el que realizar la operación.

`filedes` Descriptor de archivo sobre el que realizar la operación.

`buf` Estructura en la que la llamada guardará los datos.

El valor devuelto es igual a 0 en caso de éxito o -1 en caso de fallo. Se recomienda consultar la página del manual diferenciar la funcionalidad de cada una de ellas así como para encontrar información adicional acerca de los campos de la estructura **`struct stat`** y su significado.

2.3 Llamadas al sistema relacionadas con tiempo

2.3.1 Introducción

En esta práctica únicamente necesitamos conocer el instante de tiempo actual para realizar medidas, por lo que únicamente emplearemos la llamada al sistema **`gettimeofday`**.

2.3.2 `gettimeofday`

La llamada al sistema **`gettimeofday`** permite obtener el instante de tiempo actual en segundos y microsegundos desde el 1 de Enero de 1970. Para ello se le debe pasar por referencia una estructura de tipo **`struct timeval`** en donde almacena dicha información. Se recomienda usar la página del manual correspondiente para obtener mayor información al respecto. La declaración de la función es la siguiente:

```
int gettimeofday (struct timeval *tv, struct timezone *tz);
```

donde:

`tv` Estructura donde se devolverá el tiempo actual.

`tz` Zona horaria, o **`NULL`** si no se desea especificar.

El valor devuelto es igual a 0 en caso de éxito o -1 en caso de fallo.

2.4 Llamadas al sistema relacionadas con proyección de archivos en memoria

2.4.1 Introducción

La proyección (o *"mapeo"*) de archivos en memoria es una técnica consistente en asociar un rango de direcciones virtuales de memoria del proceso a un archivo

especificado, de forma que cuando el proceso accede a una posición de memoria en este rango de direcciones, los accesos son traducidos automáticamente por el sistema operativo a lecturas o escrituras en el archivo de forma transparente. Cuando se deshace la proyección, el sistema operativo actualiza los datos en el archivo si es necesario.

Sin embargo, proyectar el archivo no significa que éste se cargue en memoria inmediatamente; sólo cuando se realiza un acceso a una posición concreta de memoria del rango de direcciones asociado, el sistema operativo carga un bloque del archivo (típicamente una página) en dichas posiciones.

De esta forma se consiguen principalmente dos objetivos: Sólo se leen de disco aquellas partes del archivo que realmente se usan y se minimiza el número de llamadas al sistema necesarias (pues sustituimos todos los **`read`** y **`write`** por accesos a memoria).

Para trabajar con este mecanismo es necesario primero abrir el archivo con **`open`**, y a continuación realizar la llamada al sistema **`mmap`**. Finalmente se realizará un **`munmap`** y un **`close`** del archivo.

2.4.2 `mmap`

La llamada permite proyectar un archivo o parte de éste en un rango de direcciones del proceso. La declaración de la función es la siguiente:

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

donde:

`start`	Dirección sugerida de comienzo del rango, o `NULL` para dejar libertad al sistema operativo para colocar el rango asociado.
`length`	Tamaño de la zona proyectada.
`prot`	Modos de protección de la zona. Ver manual.
`flags`	*Flags* sobre el comportamiento de la proyección. Ver manual.
`fd`	Descriptor del archivo proyectado.
`offset`	Desplazamiento dentro del archivo a partir del cual se desea proyectar éste.

El valor devuelto es la dirección del rango sobre el que se ha proyectado el archivo, o `NULL` en caso de fallo.

2.4.3 `munmap`

La llamada permite deshacer la proyección de un archivo proyectado con **`mmap`**.

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

donde:

`start`	Dirección de comienzo del rango.

`length` Tamaño de la zona proyectada.

El valor devuelto 0 en caso de éxito o -1 en caso de fallo.

3 Tareas que hay que llevar a cabo

3.1 Introducción

Para esta práctica el alumno deberá realizar un programa que, empleando las llamadas al sistema anteriormente presentadas, realice accesos sobre un archivo binario proporcionado, lo modifique y lo guarde, midiendo el tiempo empleado en hacerlo. A continuación se describen las tareas concretas que hay que realizar.

3.2 Desarrollo de la práctica

Se tiene un archivo binario que contiene un número relativamente elevado de datos (ficticios) sobre alumnos matriculados en una asignatura. Dicho archivo consiste únicamente en una sucesión de estructuras de tipo **`struct evaluacion`**, cuya definición en C es la siguiente:

```
struct evaluacion
{
        char    id[16];             /* DNI o similar     */
        char    apellido1[32];
        char    apellido2[32];
        char    nombre[32];
        float   nota1p;             /* Nota 1er parcial  */
        float   nota2p;             /* Nota 2do parcial  */
        float   notamedia;          /* Nota media        */
        char    photofilename[20];  /* Nombre del JPEG   */
        int     photosize;          /* Tamaño del JPEG   */
        char    photodata[16000];   /* Datos del JPEG   */
};
```

Dado que cada estructura es relativamente grande y que hay un número grande de ellas, el archivo de datos tiene un tamaño superior a 3 Mbytes. Este archivo, llamado **`datos.bin`**, es con el que se realizarán las pruebas, y se puede descargar comprimido de la web de la asignatura.

La operación que se desea realizar es recorrer el archivo y a todo alumno con una nota media mayor o igual que 4.5 pero menor que 5, subirle la nota a 5. Esta

operación se realizará de tres formas diferentes:

- Empleando E/S clásica, con **read** y **write**, pero leyendo y escribiendo sólo una estructura del archivo de cada vez.
- Empleando E/S clásica, pero en esta ocasión primero se leerá la totalidad del archivo en un *array* de memoria reservada dinámicamente con **malloc**, se operará sobre memoria y finalmente se escribirá el *array* de datos de nuevo a disco. No está permitido realizar la práctica con un *array* estático.
- Empleando proyección de archivos en memoria. Se proyectará el archivo en memoria, se trabajará sobre ésta y a continuación se desproyectará el archivo.

Los tres programas realizan la misma tarea pero de diferente forma; de manera que las tres soluciones tendrán una parte común muy importante y una función que será la que realice el trabajo descrito de una u otra forma.

Las tres versiones diferentes de dicha función se implementarán en archivos fuente separados, cada uno de los cuales tendrá definida la función que realiza el trabajo descrito con el siguiente prototipo:

```
int revisanotas (int fd);
```

donde **fd** es el descriptor del archivo, ya abierto, sobre el que se debe trabajar, y el valor de retorno es el número de notas modificadas en la operación. Cada archivo fuente se compilará por separado generando un archivo objeto distinto, y en función de qué archivo objeto se enlace con el programa principal (que será único), se empleará un método u otro de operación.

El alumno medirá el tiempo de ejecución (en microsegundos) de las tres soluciones, empleando para ello llamadas al sistema **gettimeofday**; una de ellas justo antes de empezar a trabajar con el archivo (después de **open**) y otra justo después (antes de **close**). El código fuente correspondiente a la medición en el programa principal será por lo tanto similar al siguiente:

```
    [...]
gettimeofday( &t1, NULL);
modificados = revisanotas (fd);
gettimeofday( &t2, NULL);
    [...]
```

3.3 Realización

La práctica se realizará dividiendo el código adecuadamente en los diferentes archivos fuente necesarios y generando los archivos de cabecera **.h** correspondientes. Además, se creará obligatoriamente un archivo **Makefile** adecuado que permita recompilar los tres programas de prueba de forma automática y sin que cada vez que se realiza una modificación en un archivo fuente sea necesario recompilar y enlazar más que lo que sea necesario.

En resumen, el código que debe implementar el alumno debe quedar dividido en los siguientes archivos fuente:

practica2.c	Contendrá el programa principal **main()**, que abrirá el archivo de datos, realizará la llamada a **revisanotas()** midiendo los tiempos de ejecución y mostrará por pantalla el número de notas modificadas y el tiempo de ejecución en microsegundos.
revisanotas_std.c	Contendrá la implementación de **revisanotas()** que utiliza E/S clásica para leer y escribir estructuras de una en una.
revisanotas_malloc.c	Contendrá la implementación de **revisanotas()** que utiliza E/S clásica para leer y escribir todo el archivo en memoria de una vez, y operar desde memoria.
revisanotas_mmap.c	Contendrá la implementación de **revisanotas()** que proyecta el archivo en la memoria del proceso y opera desde allí.
practica2.h	Contendrá la declaración de la estructura **struct evaluacion** y la declaración del prototipo de la función **revisanotas()**.

La invocación de **make** sin argumentos dará como resultado tres programas ejecutables: **practica2_std**, **practica2_malloc** y **practica2_mmap**, resultado de enlazar el código objeto de **practica2.o** con cada uno de los códigos objeto resultantes de las tres implementaciones de **revisanotas()**.

La salida por pantalla de cada uno de los tres programas deberá ser similar a la siguiente:

```
user@host:~/$ ./practica2_std
Notas modificadas:    32
Tiempo empleado:      3420us
```

Es importante mencionar que no se permite el uso de variables globales en ninguna parte del programa. La no observación de esta regla supondrá que la práctica se calificará como suspensa.

El **Makefile**, además, deberá tener dos *targets* adicionales que deberán ser implementados obligatoriamente por el alumno:

clean	Borrará los ejecutables y los archivos objeto intermedios de la compilación.
datos	Descomprimirá automáticamente el archivo de datos comprimido **datos.tgz** que se descarga de la página web de la asignatura, y que residirá en el mismo directorio donde se realiza la práctica. Esto será útil ya que cada vez que se ejecute la práctica dicho archivo de datos quedará modificado y será necesario volver a recuperarlo del archivo comprimido.

Toda la práctica se evaluará **usando exclusivamente la línea de órdenes**, tanto para editar como para compilar, descomprimir el archivo de datos o ejecutar los programas, por lo que la implementación de los anteriores *targets* es obligatoria.

Como referencia para el alumno, se presentan a continuación los tiempos obtenidos (en microsegundos) para cada una de las tres realizaciones en varias arquitecturas de referencia. Se recomienda realizar varias veces cada *test* para obtener resultados homogéneos que no se vean afectados por el resto de actividad del sistema. Recordar una vez más que tras cada ejecución es necesario descomprimir de nuevo el archivo **`datos.bin`** para partir de las mismas condiciones y que se hagan las mismas modificaciones, que serán siempre 32 por las características del archivo proporcionado.

	`practica2_std`	`practica2_malloc`	`practica2_mmap`
Intel Pentium II 300Mhz	37350	121400	945
AMD Athlon 1Ghz	9300	29000	625
AMD AthlonXP 2200+ (1.8Ghz)	6260	23500	285
Intel Pentium-M 1.6Ghz	3800	16800	275
AMD Athlon64 3000+ (1.8Ghz)	2360	10320	340

4 Cuestiones adicionales

- Justifique las diferencias de tiempo de ejecución entre las tres formas de resolver el problema planteado.
- Genere y observe el programa fuente en lenguaje ensamblador correspondiente a alguno de los programas fuente escritos en lenguaje C durante la práctica.
- Utilizando la orden `ltrace` determine qué llamadas a procedimientos de biblioteca son realizados por cada una de las tres aplicaciones durante su ejecución. ¿Qué observa?
- Utilizando la orden `strace` determine qué llamadas al sistema son realizados por cada una de las tres aplicaciones durante su ejecución. Observe las diferencias y similitudes en la salida respecto al caso anterior y valore la utilidad de esta orden a la hora de depurar e investigar el comportamiento de programas.
- Elija uno de los ejecutables y observe qué tamaño tiene. Ejecutar la orden `strip` sobre él, ¿cuál es el nuevo tamaño del archivo? ¿Qué ha ocurrido?
- Si a la hora de enlazar alguno de los ejecutables se emplea la opción `-static`, ¿qué tamaño tiene el archivo ejecutable? ¿Por qué?

5 Evaluación de la práctica

La práctica se evaluará sobre 10 puntos en función de los hitos que, incrementalmente, el alumno alcance de los propuestos a lo largo del enunciado **y de la calidad del trabajo realizado**, de la siguiente forma:

- Para obtener hasta 5 puntos, el alumno realizará **completamente acorde a las especificaciones** del enunciado los archivos fuente correspondientes al **`Makefile`** (con todos los *targets*) y los dos programas que utilizan E/S estándar (**`practica2_std`** y **`practica2_malloc`**). Los programas se compilarán utilizando el modificador **`-Wall`** de `gcc` sin mostrar ningún *warning*, funcionarán correctamente y sin producir resultados incorrectos (como tiempos negativos) en ningún caso.
- Para obtener hasta 8 puntos, el alumno realizará además el programa que utiliza **`mmap()`** y tendrá el **`Makefile`** ampliado para su generación automática.
- Para obtener hasta 10 puntos, el alumno tendrá que demostrar haber realizado las cuestiones adicionales y tener claros los conceptos que se ejercita