ULPGC
Universidad de
Las Palmas de
Gran Canaria

Fundamentos de los Sistemas Operativos

Práctica 2: Archivos

El objetivo fundamental de este bloque es aprender a usar llamadas al sistema relacionadas con el sistema de archivos.

Las llamadas al sistema son la API (*Application Programming Interface*) que usan los programas para solicitar servicios al sistema operativo. Esta práctica servirá para que aprendas a usar llamadas al sistema Linux. Concretamente, trabajarás con llamadas al sistema relacionadas con archivos.

Otro objetivo de esta práctica es que aprendas a tratar las opciones y argumentos de un programa.

Recuerda que en el Campus Virtual tienes el material de apoyo para realizar esta práctica.

1 Introducción

El objetivo principal de esta práctica es que aprendas a usar llamadas al sistema Linux. Las llamadas al sistema son la API (*Application Programming Interface*) que usan los programas para solicitar servicios al sistema operativo. Las llamadas al sistema hechas en C tienen el mismo aspecto que las llamadas a funciones: se invocan con un nombre, cero o más argumentos, devuelven un resultado y probablemente tienen efectos colaterales. En esta práctica trabajarás principalmente con llamadas al sistema relacionadas con archivos.

Otro objetivo importante de esta práctica es que aprendas a tratar las opciones y argumentos que se le pasan a un programa desde la línea de órdenes.

Los objetivos de aprendizaje de esta práctica son:

- Conocer y entender cómo realizar llamadas al sistema Linux desde C.
- Tratar adecuadamente los errores que se puedan producir en las llamadas al sistema.
- Saber crear, abrir, leer, escribir y cerrar ficheros.
- Saber cómo obtener información acerca de ficheros: tipo, tamaño, permisos, fecha de modificación/creación…
- Saber utilizar los argumentos que se le pasan a un programa desde la línea de órdenes.
- Saber identificar y tratar las opciones que se le pasan a un programa desde la línea de órdenes.

Esta práctica tiene asociada una entrega y por tanto forma parte de la calificación final.

2 Requisitos previos

Para abordar esta práctica debes haber completado la Práctica 1, ya que necesitas un conocimiento mínimo de lenguaje C.

3 Plan de actividades y orientaciones

Para realizar esta práctica usaremos el siguiente material bibliográfico, en inglés:

- [1] *Linux System Programming*, *2nd Edition*. Para realizar esta práctica te vamos a proponer que consultes algunas secciones de los capítulos 1, 2 y 8. El capítulo 2 trata sobre el uso de llamadas al sistema relacionadas con archivos. La sección *Error Handling* del final del capítulo 1 es útil para aprender a tratar los posibles errores que se produzcan al invocar a las llamadas al sistema. Por último, en el capítulo 8, podrás aprender cómo usar llamadas al sistema relacionadas con permisos de ficheros y con obtención de información de estado de ficheros (*file status*).

- [2] https://people.cs.rutgers.edu/~pxk/416/notes/c-tutorials/getopt.html. Es un tutorial para aprender a recopilar opciones y argumentos desde la línea de órdenes.

Actividades	*Objetivos / orientaciones*
Leer la documentación sobre llamadas al sistema en C	Conocer los aspectos técnicos del uso de llamadas al sistema desde un programa en C, concretamente las relacionadas con manejo de archivos. Antes de empezar a resolver los problemas propuestos deberías leer la sección *System Calls* del capítulo 1 de [1].
Leer la documentación sobre tratamiento de la línea de órdenes	Conocer cómo obtener argumentos y opciones de la línea de órdenes desde un programa en C [2].
Sesiones prácticas	Habrá tres sesiones prácticas en el laboratorio. El profesor realizará ejemplos de uso de llamadas al sistema relacionadas con ficheros, gestión de errores devueltos por las llamadas al sistema y tratamiento de la línea de órdenes.
Ejercicios de entrenamiento	Es muy importante que realices los ejercicios propuestos en esta ficha. Están diseñados para aprender los aspectos esenciales del uso de llamadas al sistema. Además, puedes usarlos para resolver el ejercicio entregable.
Ejercicio entregable	Debes realizar la tarea propuesta en esta ficha y entregarla en Moodle. Te servirá para adquirir destreza en el uso de llamadas al sistema.

4 Ejercicios de entrenamiento

En este apartado te proponemos una hoja de ruta para que te adiestres de forma autónoma en los objetivos de la práctica. Te recomendamos que hagas todos los ejercicios en el orden en el que están propuestos.

Abrir y cerrar ficheros – básico

Como primer ejercicio te proponemos que crees un programa en C que intente abrir un fichero solo para leerlo e informe de si ha podido realizar la apertura o no. En caso de éxito, debe cerrar el fichero y terminar. En caso de que no se haya podido abrir el fichero, debe informarse del motivo usando el canal de salida estándar de error (*stderr*). El nombre del fichero se tomará de una cadena de caracteres contenida en el propio programa.

Para resolver este ejercicio necesitarás usar, al menos, las siguientes llamadas al sistema *open()* y *close()* (ver capítulo 2 de [1]). También deberás hacer uso de las funciones *perror()* y/o *str_error_r()* y de la variable predefinida *errno*; no olvides incluir el fichero *errno.h* en tu programa (ver sección *Error Handling* , hacia el final del capítulo 1 de [1]).

Te recomendamos que implementes este ejercicio como una función dentro de un programa C. Esta función puede recibir como parámetro la cadena con el nombre del fichero y debe ser invocada desde el programa principal; la función puede devolver un identificador de fichero válido (un número entero positivo) en caso de éxito y -1 en caso de error en la apertura del fichero.

Presta especial atención al tratamiento de los errores que puedan producirse en las llamadas al sistema. Haz que tu programa sea robusto.

Prueba tu programa con ficheros que existan y que puedas leer (por ejemplo, **/etc/passwd**, **/bin/ls**…), con ficheros que existan pero que tú no puedas leer (por ejemplo, **/var/log/messages**), con nombres de ficheros que no existan y con nombres de directorios (por ejemplo, **/etc**, **/bin**…). Comprueba que los resultados de tu programa son los esperados.

Abrir y cerrar ficheros (2)

Crea un programa con las mismas funcionalidades que el del apartado anterior, con la diferencia de que se le pasará por la línea de órdenes una lista de rutas que se intentarán abrir (ver [2]).

Tu programa debe comprobar que se le pasa al menos un argumento. Si no fuera así, debe informar por la salida estándar de error (*stderr*) y terminar. Cada argumento de la línea de órdenes se entenderá que es un nombre de fichero y se intentará abrir solo para leerlo. Si la apertura no tiene éxito, debe informarse del motivo por el canal estándar de error (*stderr*) y continuar con el siguiente nombre del fichero. Si la apertura tiene éxito, debe informarse con un breve mensaje que contenga el nombre del fichero, cerrar el fichero y continuar con el siguiente nombre.

Diseña y ejecuta una batería de pruebas para verificar que tu programa funciona correctamente. Invócalo sin argumentos, con un solo argumento, con más de un argumento…

Copiar un fichero – básico

Crea una función en C que sirva para copiar un fichero fuente en otro fichero destino. El primer argumento de la función debe ser la ruta del fichero fuente; el segundo argumento debe ser la ruta del fichero destino. Si el fichero destino existe, debe ser truncado y sobrescrito; si no existe, debe ser creado. Si por algún motivo el fichero fuente no puede abrirse o el fichero destino no puede ser escrito hay que informar del problema concreto por la salida de error estándar (*stderr*) y devolver un -1.

Para realizar este ejercicio necesitarás usar, entre otras, las llamadas al sistema *read()* y *write()*. La idea es ir leyendo bloques de bytes desde el fichero fuente e irlos escribiendo al fichero destino. Puedes consultar detalles acerca de las llamadas al sistema *read()* y *write()* en el capítulo 2 de [1] y en el manual en línea del sistema (**man 2 read**, **man 2 write**).

Asegúrate de que tu función realiza la copia correctamente. Una comprobación mínima es que los tamaños de los ficheros fuente y destino deben coincidir. También puedes usar en la consola la orden **diff** para averiguar si el contenido de ambos ficheros es idéntico. Realiza todas las pruebas necesarias para asegurarte de que tu programa funciona correctamente.

Sobreescribir un fichero

Crea un programa que sea capaz de distinguir si en la línea de órdenes se le ha pasado la opción **-f** o no. Después, modifica la función de copia de ficheros creada en el problema anterior, de forma que si el fichero destino ya existe y el programa ha sido invocado con la opción **-f**, el fichero se sobreescriba sin más. Si, por el contrario, el programa ha sido invocado sin la opción **-f**, la función debe preguntar al usuario si desea sobreescribir el fichero y actuar en consecuencia.

Para obtener una respuesta del usuario puedes usar la función *getchar()* (**man getchar**).

Obtener información de estado de un fichero

En ocasiones es necesario obtener información relacionada con el estado de un fichero (*file status*). Esto incluye, entre otras cosas, el tipo de fichero (fichero regular, directorio, enlace simbólico…), los permisos, el UID del propietario, el GID del grupo propietario, el dispositivo en el que se encuentra almacenado el fichero y el número de i-nodo[1]. Esta información se puede obtener desde el *shell* con la orden **stat**, pero vamos a pedirte que uses llamadas al sistema para averiguarla desde un programa en C.

[1] El i-nodo es una estructura de datos que el sistema operativo utiliza para gestionar los objetos del sistema de ficheros (http://es.wikipedia.org/wiki/Inodo) .

La familia de llamadas al sistema *stat()*, *fstat()* y *lstat()* devuelve información de estado de objetos del sistema de archivos. La información de estado se almacena en una estructura de datos *struct stat* (man fstat). Consulta la sección *The Stat Family* del capítulo 8 de [1] para conocer más detalles.

Te proponemos que realices los siguientes ejercicios:

- Crea una función que tome como parámetro un nombre de fichero, obtenga su información de estado y escriba por pantalla el tipo de fichero, el tamaño en bytes y el *UID* y el *GID* del propietario.
- Crea una función que tome como parámetro dos rutas y compruebe si ambos nombres se refieren al mismo objeto del sistema de ficheros. Si el número de dispositivo y el número de i-nodo coinciden, entonces se trata del mismo objeto. *Nota: puedes probar esta función comparando la ruta del directorio padre ".." con su ruta absoluta, pues se trata del mismo objeto.*

Cambiar los permisos de un fichero

Crea una función que reciba como parámetros dos nombres de ficheros, obtenga los permisos que tiene el primer fichero y asigne los mismos permisos al segundo fichero, de forma que tras invocar a la función ambos ficheros tengan idénticos permisos.

Puedes usar la familia de llamadas al sistema *stat()*, *fstat()* y *lstat()* para obtener los permisos de un fichero y la llamada al sistema *chmod(), fchmod()* (man fchmod). En la sección *Permissions* del capítulo 8 de [1] tienes información acerca de estas últimas.

5 Ejercicio entregable

El objetivo de esta práctica consiste en evolucionar la API que has desarrollado en la práctica anterior, de forma que la información sobre el estado de ocupación de la sala de teatro persista en un fichero. De esta manera siempre podremos recuperar y actualizar el estado de ocupación de la sala más allá del tiempo de vida de los programas ejecutables que hemos creado en la práctica previa.

Concretamente, deberás añadir una serie de operaciones para guardar y recuperar el estado de la sala:

Función	Comportamiento
`int guarda_estado_sala (const char* ruta_fichero);`	Guarda el estado actual de la sala en el fichero que se pasa como argumento, sobreescribiendo toda la información contenida en el fichero. Devuelve 0 si todo ha ido bien o -1 en caso de error.
`int recupera_estado_sala (const char* ruta_fichero);`	Recupera el estado de la sala a partir de la información guardada en el fichero que se pasa como argumento, sobreescribiendo toda la información del estado actual de la sala, cuyo estado actual pasará a coincidir con el contenido en el fichero. Devuelve 0 si todo ha ido bien o -1 en caso de error.

`int guarda_estadoparcial_sala (const char* ruta_fichero, size_t num_asientos, int* id_asientos);`	Guarda el estado de un conjunto de asientos de la sala en un fichero preexistente que se pasa como argumento, sobreescribiendo la información contenida en el fichero. Devuelve 0 si todo ha ido bien o -1 en caso de error.
`int recupera_estadoparcial_sala (const char* ruta_fichero, size_t num_asientos, int* id_asientos);`	Recupera el estado de un conjunto de asientos de la sala a partir de la información guardada en un fichero preexistente que se pasa como argumento, sobreescribiendo la información del estado actual del conjunto de asientos que se pasa por parámetro, cuyo estado actual pasará a coincidir con el contenido en el fichero. Devuelve 0 si todo ha ido bien o -1 en caso de error.

Además, debes crear un programa llamado **misala** que permita realizar diversas operaciones sobre los asientos de una sala. El programa **misala** se invocará según esta sintaxis general:

`misala` *`orden argumentos`*

Las diferentes órdenes que se deben implementar y sus respectivos argumentos son los siguientes:

- `misala` *`crea`* `–f[o]` *`ruta`* `–c` *`capacidad`*: crea una sala con la capacidad indicada y guarda el estado inicial (sala con todos sus asientos disponibles) en el fichero especificado a través del parámetro *ruta*. La capacidad de la sala se guardará también al comienzo del fichero. Si el fichero que se especifica por parámetro existe y se especifica la opción 'o', se sobreescribirá su contenido con la información de la nueva sala creada. Si el fichero existe pero no se especifica la opción 'o' , el programa terminará informando al usuario del error por *stderr*. Si la ruta no es válida o no se tienen los permisos adecuados, se imprimirá un error por *stderr*.
- `misala` *`reserva`* `-f` *`ruta`* `-n` *`número_de_asientos`* `id_persona1 id_persona2…`: realiza la reserva de un número de asientos que se indica por línea de órdenes para las personas cuyos identificadores también se especifican en la orden. El número de identificadores de persona que se indiquen en la orden deberá coincidir con el número de asientos indicados, debiéndose informar de un error por *stderr* en caso contrario, en cuyo caso no se realizará ninguna reserva. El argumento "*número_de_asientos*" debe ser un entero positivo mayor que cero. En caso contrario, se imprimirá un error por *stderr*. Si la ruta no es válida o no se tienen los permisos adecuados, se imprimirá un error por *stderr*.
- `misala` *`anula`* `-f` *`ruta`* `-a id_`*`asiento1`* `[id_`*`asiento2...`*`]`: anula las reservas de los asientos que se pasan por línea de órdenes. Los identificadores de asientos serán enteros positivos válidos dentro de la capacidad de la sala. En caso contrario, se realizarán las reservas de los identificadores que se encuentren dentro del rango válido y se imprimirá un mensaje de error por

stderr informando de los asientos no válidos. Si la ruta no es válida o no se tienen los permisos adecuados, se imprimirá un error por *stderr*.

- `misala` *`estado`* `-f` *`ruta`*: muestra por consola el estado de la sala contenido en el fichero especificado a través del parámetro *ruta.* Si la ruta no es válida o no se tienen los permisos adecuados, se imprimirá un error por *stderr*.

Implementación. Para implementar las acciones sobre ficheros solo te permitimos que uses *llamadas al sistema* de Linux, es decir, las operaciones open(), close(), creat(), read(), write(), stat(), lseek(), etc. No puedes utilizar funciones de más alto nivel, por ejemplo las de <stdio.h>. Como criterio orientativo para saber si una función es una llamada al sistema, estas se encuentran en las cabeceras **<unistd.h>** y **<fcntl.h>**.

Pruebas. Asegúrate de planificar y ejecutar una batería exhaustiva de pruebas sobre tu programa para asegurar su validez y su robustez. Asegúrate de que tu programa cumple con todos los requisitos exigidos. No permitas que haya casos en los que no funcione. Comprueba todos los resultados de las llamadas al sistema y trata adecuadamente los errores. No seas indulgente con tu código. Los profesores no lo seremos.

Entrega. Recuerda que debes entregar los ficheros fuentes que implementan la solución a este ejercicio entregable, junto con la ficha correspondiente en formato PDF. Empaqueta todos los ficheros en un archivo **tar.gz** y súbelos al Campus Virtual.

6 Criterios de evaluación

Esta práctica recibirá una calificación en la escala «superada / no superada».

Superada

Para que una entrega sea calificada como *superada* deberá cumplir **todos** estos requisitos:

- Las nuevas funciones incorporadas a la API funcionan de acuerdo con las especificaciones.
- El programa detecta los escenarios de error de acceso a los ficheros que se pasen como argumento y notifica errores en tales casos.
- El código está correctamente separado en módulos (.h, .c, etc.).
- El código supera los tests propuestos por el equipo docente.
- Se ha implementado al menos una prueba para cada función de la biblioteca.
- Se ha entregado una ficha/memoria entendible y conforme con la plantilla.
- La ficha describe cómo debe compilarse y probarse el código.
- En la ficha se documentan las pruebas que se han realizado para verificar que el código es correcto.

7 Reto

Tal y como se indica en el proyecto docente de la asignatura, cada práctica puede contener retos adicionales que contribuyen a la calificación final de la parte práctica de la asignatura. En el caso concreto de esta práctica, podrás obtener 0,7 puntos adicionales si se cumplen las siguientes dos condiciones:

1. El código cumple los siguientes requisitos:
   - **Mantenibilidad**: legibilidad del código (identificadores entendibles para variables, funciones, constantes, etc.); organización modular; reutilización de código (no hay bloques duplicados de código); uso de **getopt** para tratamiento de argumentos; facilidad de parametrización/configuración; rutinas para testeo del código, etc.
   - **Eficiencia**: implementación eficiente en cuanto a tiempo de ejecución y a consumo de recursos.
   - **Robustez**: comprobaciones de errores, mensajes de error, comprobación de límites de arrays, cuidado con las variables sin inicializar, cuidado con el uso de punteros, etc.
2. Se implementa la siguiente funcionalidad:
   - `misala` *`anula`* `-f` *`ruta`* `-i` *`id_persona1`* `[`*`id_persona2...`*`]`: anula todas las reservas de los *id* de personas que se pasan por línea de órdenes. Si la ruta no es válida o no se tienen los permisos adecuados, se imprimirá un error por *stderr*. En caso de que no se encuentren reservas para algunos identificadores de persona, la anulación tendrá efecto sobre los *id* encontrados y se imprimirá el mensaje de error correspondiente por *stderr* informando de los *id* no encontrados.
   - `misala` *`compara ruta1 ruta2`*: devuelve 0 si el estado de las salas almacenadas en *ruta1* y *ruta2* son iguales, lo que implica que ambas salas deben coincidir en tamaño y también deberán coincidir las reservas de todos sus asientos, 1 si son diferentes o -1 en caso de error. Si las rutas no son válidas o no se tienen los permisos adecuados, se imprimirá un error por *stderr*.