Prácticas de Sistemas Operativos

Módulo II. Uso de los Servicios del SO mediante la API

Sesión 2. Llamadas al sistema para el Sistema de Archivos. Parte II

1. Objetivos principales

Esta sesión está pensada para trabajar con las llamadas al sistema que modifican los permisos de un archivo y con los directorios.

- Conocer y saber usar las órdenes para poder modificar y controlar los permisos de los archivos que crea un proceso basándose en la máscara que tiene asociado el proceso.
- Conocer las funciones y estructuras de datos que nos permiten trabajar con los directorios.
- Comprender los conceptos e implementaciones que utiliza un sistema operativo UNIX para construir las abstracciones de archivos y directorios.

2. Llamadas al sistema relacionadas con los permisos de los archivos

En este punto trabajaremos ampliando la información que ya hemos visto en la sesión 1. Vamos a entender por qué cuando un proceso crea un archivo se le asignan a dicho archivo unos permisos concretos. También nos interesa controlar los permisos que queremos que tenga un archivo cuando se crea.

2.1 La llamada al sistema `umask`

La llamada al sistema `umask` fija la máscara de creación de permisos para el proceso y devuelve el valor previamente establecido. El argumento de la llamada puede formarse mediante una combinación `OR` de las nueve constantes de permisos (`rwx` para `ugo`) vistas anteriormente. A continuación se muestra un resumen de la página del manual del programador de Linux para esta llamada:

NOMBRE

umask - establece la máscara de creación de ficheros

SINOPSIS

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

mode_t umask(mode_t mask);
```

DESCRIPCIÓN

`umask` establece la máscara de usuario a mask & 0777.

La máscara de usuario es usada por `open(2)` para establecer los permisos iniciales del archivo que se va a crear. Específicamente, los permisos presentes en la máscara se desactivan del argumento `mode` de `open` (así pues, por ejemplo, si creamos un archivo con campo mode= `0666` y tenemos el valor común por defecto de `umask=022`, este archivo se creará con permisos: **`0666 & ~022 = 0644 = rw-r--r--`**, que es el caso más normal).

VALOR DEVUELTO

Esta llamada al sistema siempre tiene éxito y devuelve el valor anterior de la máscara.

2.2 Las llamadas al sistema `chmod` y `fchmod`.

Estas dos funciones nos permiten cambiar los permisos de acceso para un archivo que existe en el sistema de archivos. La llamada `chmod` sobre un archivo especificado por su pathname mientras que la función `fchmod` opera sobre un archivo que ha sido previamente abierto con `open`.

A continuación se muestra un resumen de la página del manual del programador de Linux para esta llamada:

NOMBRE

chmod, fchmod - cambia los permisos de un archivo

SINOPSIS

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

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

DESCRIPCIÓN

Cambia los permisos del archivo dado mediante `path` o referido por `fildes`. Los permisos se pueden especificar mediante un `OR` lógico de los siguientes valores:

S_ISUID	04000	activar la asignación del UID del propietario al UID efectivo del proceso que ejecute el archivo.
S_ISGID	02000	activar la asignación del GID del propietario al GID efectivo del proceso que ejecute el archivo.
S_ISVTX	01000	activar sticky bit
S_IRWXU	00700	**user** (propietario del archivo) tiene permisos de lectura, escritura y ejecución
S_IRUSR	00400	lectura para el propietario (= S_IREAD no POSIX)
S_IWUSR	00200	escritura para el propietario (= S_IWRITE no POSIX)
S_IXUSR	00100	ejecución/búsqueda para el propietario (=S_IEXEC no POSIX)
S_IRWXG	00070	**group** tiene permisos de lectura, escritura y ejecución
S_IRGRP	00040	lectura para el grupo
S_IWGRP	00020	escritura para el grupo

S_IXGRP	00010	ejecución/búsqueda para el grupo
S_IRWXO	00007	**other** tienen permisos de lectura, escritura y ejecución
S_IROTH	00004	lectura para otros
S_IWOTH	00002	escritura para otros
S_IXOTH	00001	ejecución/búsqueda para otros

VALOR DEVUELTO

En caso de éxito, devuelve 0. En caso de error, -1 y se asigna a la variable **errno** un valor adecuado.

Avanzado

Para cambiar los bits de permisos de un archivo, el UID efectivo del proceso debe ser igual al del propietario del archivo, o el proceso debe tener permisos de root o superusuario (UID efectivo del proceso debe ser 0).

Si el UID efectivo del proceso no es cero y el grupo del fichero no coincide con el ID de grupo efectivo del proceso o con uno de sus *ID's de grupo suplementarios*, el bit S_ISGID se desactivará, aunque esto no provocará que se devuelva un error.

Dependiendo del sistema de archivos, los bits S_ISUID y S_ISGID podrían desactivarse si el archivo es escrito. En algunos sistemas de archivos, solo el root puede asignar el 'sticky bit', lo cual puede tener un significado especial (por ejemplo, para directorios, un archivo sólo puede ser borrado por el propietario o el root).

Actividad 2.1 Trabajo con llamadas de cambio de permisos

Consulta el manual en línea para las llamadas al sistema `umask` y `chmod`.

Ejercicio 1. ¿Qué hace el siguiente programa?

```
/*
tarea3.c

Trabajo con llamadas al sistema del Sistema de Archivos ''POSIX 2.10
compliant''

Este programa fuente está pensado para que se cree primero un programa con la
parte de CREACION DE ARCHIVOS y se haga un ls -l para fijarnos en los permisos
y entender la llamada umask.
En segundo lugar (una vez creados los archivos) hay que crear un segundo
programa con la parte de CAMBIO DE PERMISOS para comprender el cambio de
permisos relativo a los permisos que actualmente tiene un archivo frente a un
establecimiento de permisos absoluto.
*/

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

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

int main(int argc, char *argv[])
{
int fd1,fd2;
struct stat atributos;

//CREACION DE ARCHIVOS
if( (fd1=open("archivo1",O_CREAT|O_TRUNC|O_WRONLY,S_IRGRP|S_IWGRP|S_IXGRP))<0)
{
      printf("\nError %d en open(archivo1,...)",errno);
      Sistemas Operativos II Guión de Prácticas, pág.12
      perror("\nError en open");
      exit(-1);
}
umask(0);
if( (fd2=open("archivo2",O_CREAT|O_TRUNC|O_WRONLY,S_IRGRP|S_IWGRP|S_IXGRP))<0)
{
      printf("\nError %d en open(archivo2,...)",errno);
      perror("\nError en open");
      exit(-1);
}

//CAMBIO DE PERMISOS
if(stat("archivo1",&atributos) < 0) {
      printf("\nError al intentar acceder a los atributos de archivo1");
      perror("\nError en lstat");
      exit(-1);
}
if(chmod("archivo1", (atributos.st_mode & ~S_IXGRP) | S_ISGID) < 0) {
      perror("\nError en chmod para archivo1");
      exit(-1);
}
if(chmod("archivo2",S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH) < 0) {
      perror("\nError en chmod para archivo2");
      exit(-1);
}
close(fd1);
close(fd2);

return 0;
}
```

3. Funciones de manejo de directorios

Aunque los directorios se pueden leer utilizando las mismas llamadas al sistema que para los archivos normales, como la estructura de los directorios puede cambiar de un sistema a otro, los programas en este caso no serían transportables. Para solucionar este problema, se va a utilizar una biblioteca estándar de funciones de manejo de directorios que se presentan de forma resumida a continuación:

· **opendir**: se le pasa el `path` del directorio a abrir, y devuelve un puntero a la estructura de tipo *`DIR`*, llamada *stream* de directorio. El tipo *`DIR`* está definido en *`<dirent.h>`*.

· **readdir**: lee la entrada donde esté situado el puntero de lectura de un directorio ya abierto cuyo *stream* se pasa a la función. Después de la lectura adelanta el puntero una posición. Devuelve la entrada leída a través de un puntero a una estructura (`struct dirent`), o devuelve `NULL` si llega al final del directorio o se produce un error.

· **closedir**: cierra un directorio, devolviendo **0** si tiene éxito, en caso contrario devuelve **-1**.

· **seekdir**: permite situar el puntero de lectura de un directorio (se tiene que usar en combinación con `telldir`).

· **telldir**: devuelve la posición del puntero de lectura de un directorio.

· **rewinddir**: posiciona el puntero de lectura al principio del directorio.

A continuación se dan las declaraciones de estas funciones y de las estructuras que se utilizan, contenidas en los archivos `<sys/types.h>` y `<dirent.h>` y del tipo *`DIR`* y la estructura *`dirent`* (entrada de directorio).

```
DIR *opendir(char *dirname)
struct dirent *readdir(DIR *dirp)
int closedir(DIR *dirp)
void seekdir(DIR *dirp, log loc)
long telldir(DIR *dirp)
void rewinddir(DIR *dirp)

typedef struct _dirdesc {
      int dd_fd;
      long dd_loc;
      long dd_size;
      long dd_bbase;
      long dd_entno;
      long dd_bsize;
      char *dd_buf;
} DIR;


//La estructura struct dirent conforme a POSIX 2.1 es la siguiente:
#include <sys/types.h>
#include <dirent.h>
struct dirent {
      long d_ino; /* número i-nodo */
      off_t d_off; /* desplazamiento al siguiente dirent */
      unsigned short d_reclen; /* longitud de esta entrada */
      unsigned char d_type; /* tipo de archivo*/
      char d_name[256]; /* nombre del archivo */
};
```

Actividad 3.1 Trabajo con funciones estándar de manejo de directorios

Mirad las funciones estándar de trabajo con directorios utilizando `man opendir` y viendo el resto de funciones que aparecen en la sección VEASE TAMBIEN de esta página del manual.

Ejercicio 2. Realiza un programa en C utilizando las llamadas al sistema necesarias que acepte como entrada:

- Un argumento que representa el '***pathname***' de un directorio.

- Otro argumento que es un ***número octal de 4 dígitos*** (similar al que se puede utilizar para cambiar los permisos en la llamada al sistema `chmod`). Para convertir este argumento tipo cadena a un tipo numérico puedes utilizar la función `strtol`. Consulta el manual en línea para conocer sus argumentos.

El programa tiene que usar el número octal indicado en el segundo argumento para cambiar los permisos de todos los archivos que se encuentren en el directorio indicado en el primer argumento.
El programa debe proporcionar en la salida estándar una línea para cada archivo del directorio que esté formada por:

```
<nombre_de_archivo> : <permisos_antiguos> <permisos_nuevos>
```

Si no se pueden cambiar los permisos de un determinado archivo se debe especificar la siguiente información en la línea de salida:

```
<nombre_de_archivo> : <errno> <permisos_antiguos>
```

Ejercicio 3. Programa una nueva orden que recorra la jerarquía de subdirectorios existentes a partir de uno dado como argumento y devuelva la cuenta de todos aquellos archivos regulares que tengan permiso de ejecución para el *grupo* y para *otros*. Además del nombre de los archivos encontrados, deberá devolver sus números de inodo y la suma total de espacio ocupado por dichos archivos. El formato de la nueva orden será:

```
$ ./buscar <pathname>
```

donde **<***pathname***>** especifica la ruta del directorio a partir del cual queremos que empiece a analizar la estructura del árbol de subdirectorios. En caso de que no se le de argumento, tomará como punto de partida el *directorio actual*. Ejemplo de la salida después de ejecutar el programa:

```
Los i-nodos son:

./a.out 55

./bin/ej 123

./bin/ej2 87

...

Existen 24 archivos regulares con permiso x para grupo y otros

El tamaño total ocupado por dichos archivos es 2345674 bytes
```

3.1 Recorriendo el sistema de archivos usando `nftw`

Si bien con las funciones anteriores podemos recorrer el sistema de archivos de forma “manual”, la función `nftw()` permite recorrer recursivamente un sub-árbol y realizar alguna operación sobre los archivos del mismo.

```
#define _XOPEN_SOURCE 500

#include <ftw.h>


int nftw(const char *dirpath,

            int (*func) (const char *pathname, const struct stat *statbuf,

            int typeflag, struct FTW *ftwbuf),

            int nopenfd, int flags);


                  Retorna: 0 si recorre completamente el árbol; -1 si error
o el primer valor no cero devuelto por func.
```

La función recorre el árbol de directorios especificado por `dirpath` y llama a la función `func` definida por el programador para cada archivo del árbol. Por defecto, `ntfw` realiza un recorrido no ordenado en preorden del árbol, procesando primero cada directorio antes de procesar los archivos y subdirectorios dentro del directorio.

Mientras se recorre el árbol, la función `nftw` abre al menos un descriptor de archivo por nivel del árbol. El parámetro `nopenfd` especifica el máximo número de descriptores que puede usar. Si la profundidad del árbol es mayor que el número de descriptores, la función evitar abrir más cerrando y reabriendo descriptores.

El argumento `flags` es creado mediante un OR (|) con cero o más contantes, que modifican la operación de la función:

`FTW_DIR` Realiza un `chdir` (cambia de directorio) en cada directorio antes de procesar su contenido. Se utiliza cuando `func` debe realizar algún trabajo en el directorio en el que el archivo especificado por su argumento `pathname` reside.

`FTW_DEPTH` Realiza un recorrido postorden del árbol. Esto significa que `nftw` llama a `func` sobre todos los archivos (y subdirectorios) dentro del directorio antes de ejecutar func sobre el propio directorio.

`FTW_MOUNT` No cruza un punto de montaje.

`FTW_PHYS` Indica a `nftw` que nos desreferencie los enlaces simbólicos. En su lugar, un enlace simbólico se pasa a `func` como un valor `typedflag` de `FTW_SL`.

Para cada archivo, `nftw()` pasa cuatro argumentos al invocar a `func`. El primero, `pathname` indica el nombre del archivo, que puede ser absoluto o relativo dependiendo de si `dirpath` es de un tipo u otro. El argumento `statbuf` es un puntero a una estructura `stat` conteniendo información del archivo. El tercer argumento, `typeflag`, suministra información adicional sobre el archivo, y tiene uno de los siguientes nombres simbólicos:

`FTW_D` Es un directorio

`FTW_DNR` Es un directorio que no puede leerse (no se leen sus descendientes).

`FTW_DP` Estamos haciendo un recorrido postorden de un directorio, y el ítem actual es un directorio cuyos archivos y subdirectorios han sido recorridos.

`FTW_F` Es un archivo de cualquier tipo diferente de un directorio o enlace simbólico.

`FTW_NS` `stat` ha fallado sobre este archivo, probablemente debido a restricciones de permisos. El valor `statbuf` es indefinido.

`FTW_SL` Es un enlace simbólico. Este valor se retorno solo si nftw se invoca con `FTW_PHYS`.

`FTW_SLN` El ítem es un enlace simbólico perdido. Este se da cuando no se especifica `FTW_PHYS`.

El cuarto elemento de `func` es `ftwbuf`, es decir, un puntero a una estructura que se define de la forma:

```
struct FTW {

      int base;   /* Desplazamiento de la parte base del pathname */

      int level;  /*Profundidad del archivo dentro recorrido del arbol */

};
```

El campo base es un desplazamiento entero de la parte del nombre del archivo (el componente después del último /) del `pathname` pasado a `func`.

Cada vez que es invocada, `func` debe retornar un valor entero que es interpretado por `nftw`. Si retorna 0, indica a `nftw` que continúe el recorrido del árbol. Si todas las invocaciones a `func` retornan cero, `nftw` retornará 0. Si retorna distinto de cero, `nftw` para inmediatamente, en cuyo caso `nftw` retorna este valor como su valor de retorno.

El Programa 3 muestra un ejemplo de uso de la función. En este caso, utilizamos `nftw` para recorrer el directorio pasado como argumento, salvo que no se especifique, en cuyo caso, actuamos sobre el directorio actual. Para cada elemento atravesado se invoca a la función `visitar()` que imprime el `pathname` y el modo en octal. Observar que para que el recorrido sea completo la función visitar debe devolver un 0, sino se detendría la búsqueda.

Programa 3.- Recorrer un sub-árbol con la función nftw.

```
#include <iostream>
#include <stdio.h>

#include <ftw.h>


using namespace std;

int visitar(const char* path, const struct stat* stat, int flags, struct FTW*
ftw) {

    cout<<"Path: "<< path <<" Modo: "<<stat->st_mode<<endl;

    return 0;

}

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

    if (nftw(argc >= 2 ? argv[1] : ".", visitar, 10, 0) != 0) {

        perror("nftw");

    }

}
```