Práctica 2.2. Sistema de Ficheros

Objetivos

En esta práctica se revisan las funciones del sistema básicas para manejar un sistema de ficheros, referentes a la creación de ficheros y directorios, duplicación de descriptores, obtención de información de ficheros o el uso de cerrojos.

Contenidos

Preparación del entorno para la práctica
Creación y atributos de ficheros
Redirecciones y duplicación de descriptores
Cerrojos de ficheros
Directorios

Preparación del entorno para la práctica

La realización de esta práctica únicamente requiere del entorno de desarrollo (compilador, editores y utilidades de depuración). Estas herramientas están disponibles en las máquinas virtuales de la asignatura y en la máquina física de los puestos del laboratorio.

Creación y atributos de ficheros

El inodo de un fichero guarda diferentes atributos de éste, como por ejemplo el propietario, permisos de acceso, tamaño o los tiempos de acceso, modificación y creación. En esta sección veremos las llamadas al sistema más importantes para consultar y fijar estos atributos así como las herramientas del sistema para su gestión.

Ejercicio 1. `ls(1)` muestra el contenido de directorios y los atributos básicos de los ficheros. Consultar la página de manual y estudiar el uso de las opciones `-a -l -d -h -i -R -1 -F` y `--color`. Estudiar el significado de la salida en cada caso.

-a: todas las entradas aunque empiecen por .
-l: formato largo de lista
-d: listar directorios -no contenidos
-h: human readable
-i: inodo para cada archivo
-r: reverse order
-1: un archivo por linea
-F: append '/' a directorios

Ejercicio 2. El *modo* de un fichero es `<tipo><rwx_propietario><rwx_grupo><rwx_resto>`:

- `tipo`: `-` fichero ordinario; `d` directorio; `l` enlace; `c` dispositivo carácter; `b` dispositivo bloque; `p` FIFO; `s` socket
- `rwx`: `r` lectura (4); `w` escritura (2); `x` ejecución (1)

Comprobar los permisos de algunos directorios (con `ls -ld`).

Ejercicio 3. Los permisos se pueden otorgar de forma selectiva usando la notación octal o la simbólica. Ejemplo, probar las siguientes órdenes (equivalentes):

- `chmod 540 fichero`
- `chmod u+rx,g+r-wx,o-wxr fichero`

¿Cómo se podrían fijar los permisos `rw-r--r-x`, de las dos formas?

```
chmod 540 p22.txt
ls -ld
-r-xr----- 1 enrique enrique 284 oct 28 17:13 p22.txt

chmod u+rx,g+r-wx,o-wxr p22.txt
ls -ld
-r-xr----- 1 enrique enrique 284 oct 28 17:13 p22.txt
```

Respuesta a la pregunta:

```
chmod 645 p22.txt
ls -ld p22.txt
-rw-r--r-x 1 enrique enrique 284 oct 28 17:13 p22.txt

chmod u+rw-x,g+r-wx,o+rx-w p22.txt
ls -ld p22.txt
-rw-r--r-x 1 enrique enrique 284 oct 28 17:13 p22.txt
```

Ejercicio 4. Crear un directorio y quitar los permisos de ejecución para usuario, grupo y otros. Intentar cambiar al directorio.

```
chmod 000 Mierdectorio
cd Mierdectorio
bash: cd: Mierdectorio: Permission denied
```

Ejercicio 5. Escribir un programa que, usando `open(2)`, cree un fichero con los permisos `rw-r--r-x`. Comprobar el resultado y las características del fichero con `ls(1)`.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(){
        open("/home/enrique/Documents/ASOR/fichero225", O_CREAT, 0645);
        return 0;
}
```

Ejercicio 6. Cuando se crea un fichero, los permisos por defecto se derivan de la máscara de usuario (*umask*). El comando interno de la *shell* `umask` permite consultar y fijar esta máscara. Usando este comando, fijar la máscara de forma que los nuevos ficheros no tengan permiso de escritura para el grupo y no tengan ningún permiso para otros. Comprobar el funcionamiento con `touch(1)`, `mkdir(1)` y `ls(1)`.

```
umask 027
```

Ejercicio 7. Modificar el ejercicio 5 para que, antes de crear el fichero, se fije la máscara igual que en el ejercicio 6. Comprobar el resultado con `ls(1)`. Comprobar que la máscara del proceso padre (la *shell*) no cambia.

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

int main(){
        mode_t ant = umask(027);
        open("/home/enrique/Documents/ASOR/fichero225", O_CREAT, 0777);
        mode_t current = umask(prev);
        printf("Current mask: %i\n", current);
        return 0;
}
```

Ejercicio 8. `ls(1)` puede mostrar el inodo con la opción `-i`. El resto de información del inodo puede obtenerse usando `stat(1)`. Consultar las opciones del comando y comprobar su funcionamiento.

Ejercicio 9. Escribir un programa que emule el comportamiento de `stat(1)` y muestre:

- El número *major* y *minor* asociado al dispositivo.
- El número de inodo del fichero.
- El tipo de fichero (directorio, enlace simbólico o fichero ordinario).
- La hora en la que se accedió el fichero por última vez. ¿Qué diferencia hay entre `st_mtime` y `st_ctime`?

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

int main(int argc, char *argv[]){
  if(argc < 2){
    printf("Missing filename\nUsage: [PATH/filename]\n");
    return 1;
  }
  printf("\n%s\n\n", argv[1]);

  char * f = argv[1];
  struct stat buf;

  if(stat(f,&buf)!= 0){
    printf("Error locating \t[%s]\n", f);
    return 1;
  }

  printf("\tMajor number: \t%li\n", (long) minor(buf.st_dev));
        printf("\tMinor number: \t%li\n", (long) major(buf.st_dev));
  printf("\ti-Node number: \t%li\n", buf.st_ino);
  mode_t mode = buf.st_mode;
  printf("\tMode: \t%i\t", mode);
  if(S_ISLNK(mode)){
    printf("[Link]\n");
  }
  else if (S_ISREG(mode)){
```

```
    printf("[Normal file]\n");
  }
  else if (S_ISDIR(mode)){
    printf("[Directory]\n");
  }
  else printf("[Other]\n");

  time_t access_last = buf.st_atime;
  time_t modify_last = buf.st_mtime;

  struct tm * t_access_last = localtime(&access_last);
  struct tm * t_modify_last = localtime(&modify_last);

  printf("\tLast time accessed: \t%d:%d\n", t_access_last->tm_hour, t_access_last->tm_min);
  printf("\tLast time modified: \t%d:%d\n", t_modify_last->tm_hour, t_modify_last->tm_min);

  printf("\n");

  return 0;
}
```

Ejemplo de uso:

```
g++ ej229.cpp && ./a.out
Missing filename
Usage: [PATH/filename]

g++ ej229.cpp && ./a.out /home/hlocal/Documentos/ASOR/ej229.cpp

/home/hlocal/Documentos/ASOR/ej229.cpp

        Major number:   9
        Minor number:   8
        i-Node number:  1186105
        Mode:    33188   [Normal file]
        Last time accessed:         11:59
        Last time modified:         11:59



g++ ej229.cpp && ./a.out /home/hlocal/Documentos/ASOR

/home/hlocal/Documentos/ASOR

        Major number:   9
        Minor number:   8
        i-Node number:  1186103
        Mode:    16877   [Directory]
        Last time accessed:         12:3
        Last time modified:         12:3
```

Ejercicio 10. Los enlaces se crean con `ln(1)`:

- Con la opción `-s`, se crea un enlace simbólico. Crear un enlace simbólico a un fichero ordinario y otro a un directorio. Comprobar el resultado con `ls -l` y `ls -i`. Determinar el inodo de cada fichero.
- Repetir el apartado anterior con enlaces rígidos. Determinar los inodos de los ficheros y las propiedades con `stat` (observar el atributo número de enlaces).
- ¿Qué sucede cuando se borra uno de los enlaces rígidos? ¿Qué sucede si se borra uno de los enlaces simbólicos? ¿Y si se borra el fichero original?

touch ej2210
mkdir ej2210d
ls
ej2210
ln -s ej2210 ej2210link
ln -s ej2210d ej2210dlink
ls -i
1186345 ej2210
1186346 ej2210dlink
1317292 ej2210d
1186343 ej2210link

No se puede crear un hard link a un directorio

ln ej2210 ej2210h
ls
ej2210 ej2210h
ls -i
1186345 ej2210 1186345 ej2210h

Borrando enlace rígido: no pasa nada al original, se decrementa 1 el link counter, si es 0 ya no hay links al inodo, si hay simbólicos apuntando se corrompen
Borrando enlace simbólico: no pasa nada al original, se decrementa 1 el link counter

Ejercicio 11. `link(2)` y `symlink(2)` crean enlaces rígidos y simbólicos, respectivamente. Escribir un programa que reciba una ruta a un fichero como argumento. Si la ruta es un fichero regular, creará un enlace simbólico y rígido con el mismo nombre terminado en `.sym` y `.hard`, respectivamente. Comprobar el resultado con `ls(1)`.

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

int main(int argc, char *argv[]){
   if(argc < 2){
```

```
    printf("Missing filename\nUsage: [PATH/filename]\n");
    return 1;
  }
  printf("\nLinking %s\n\n",  argv[1]);

  char * f = argv[1];
  struct stat buf;

  if(stat(f,&buf)!= 0){
    printf("Error locating \t[%s]\n", f);
    return 1;
  }
  else if (!S_ISREG(buf.st_mode)){
    printf("This is not a file! \t[%s]\n", f);
    return 1;
  }

  char* f1 = (char *) malloc(sizeof(char)*(4 + strlen(f)));
        char* f2 = (char *) malloc(sizeof(char)*(5 + strlen(f)));

        strcpy(f1, argv[1]); strcat(f1, ".sym");
        strcpy(f2, argv[1]); strcat(f2, ".hard");


  if(symlink(f, f1) != 0){
    printf("Stopped: Error creating symlink \t[%s]\n", f1);
    return 1;
  }
  if(link(f, f2) != 0){
    printf("Stopped: Error creating hardlink \t[%s]\n", f2);
    return 1;
  }

  printf("\tSymlink: \t[%s]\n",  f1);
  printf("\tHardlink: \t[%s]\n",  f2);
  printf("\nSuccess!\n");

  return 0;
}
```

```
g++ ej2211.cpp && ./a.out ej2211.cpp

Linking ej2211.cpp

   Symlink:    [ej2211.cpp.sym]
   Hardlink:    [ej2211.cpp.hard]

Success!

ls
a.out     ej2211.cpp.hard  ej225.cpp  ej229.cpp
ej2211.cpp  ej2211.cpp.sym   ej227.cpp
```

Redirecciones y duplicación de descriptores

La *shell* proporciona operadores (>, >&, >>) que permiten redirigir un fichero a otro, ver los ejercicios propuestos en la práctica opcional. Esta funcionalidad se implementa mediante `dup(2)` y `dup2(2)`.

Ejercicio 12. Escribir un programa que redirija la salida estándar a un fichero cuya ruta se pasa como primer argumento. Probar haciendo que el programa escriba varias cadenas en la salida estándar.

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

int main(int argc, char *argv[]){
   if(argc < 2){
      printf("Missing filename\nUsage: [PATH/filename]\n");
      return 1;
   }

   char * f = argv[1];
   printf("\n Redirecting to %s\n\n",  f);

   int fd = open(f, O_CREAT | O_RDWR, 00777);
   if (fd == -1){
      printf("Error opening  or creating\t[%s]\n", f);
      return 1;
   }

   printf("\tPrinting \t[Hola Mundo.]\n");
   dup2(fd, 1);
   printf("Hola Mundo\n");

   printf("\nSuccess!\n");

   return 0;
}
```

Ejercicio 13. Modificar el programa anterior para que también redirija la salida estándar de error al fichero. Comprobar el funcionamiento incluyendo varias sentencias que impriman en ambos flujos. ¿Hay diferencia si las redirecciones se hacen en diferente orden? ¿Por qué `ls > dirlist 2>&1` es diferente a `ls 2>&1 > dirlist`?

```
dup2(fd, 2);
```

```
ls > dirlist 2>&1
Redirecciona salida estándar de ls a dirlist, y la salida de errores a 1, que corresponde a dirlist

ls 2>&1 > dirlist
Redirecciona salida de errores a 1 y después salida estándar de ls a dirlist
```

Cerrojos de ficheros

El sistema de ficheros ofrece cerrojos de ficheros consultivos.

Ejercicio 14. El estado y cerrojos de fichero en uso en el sistema se pueden consultar en el fichero `/proc/locks`. Estudiar el contenido de este fichero.

https://docs.fedoraproject.org/en-US/Fedora/16/html/System_Administrators_Guide/s2-proc-locks.html
Muy interesante

Ejercicio 15. Escribir un programa que consulte y muestre en pantalla el estado del cerrojo sobre un fichero usando `lockf(3)`. El programa mostrará el estado del cerrojo (bloqueado o desbloqueado). Además:

- Si está desbloqueado, fijará un cerrojo y escribirá la hora actual. Después suspenderá su ejecución durante 30 segundos (con `sleep(3)`) y a continuación liberará el cerrojo.
- Si está bloqueado, terminará el programa.

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

int main(int argc, char *argv[]){
   if(argc < 2){
      printf("Missing filename\nUsage: [PATH/filename]\n");
      return 1;
   }

   char * f = argv[1];
   printf("\nUsing [%s]\n\n",  f);

   int fd = open(f, O_CREAT | O_RDWR, 00777);
   if (fd == -1){
      printf("Error opening  or creating\t[%s]\n", f);
      return 1;
   }

   if(lockf(fd, F_TEST, 0) == -1){
      printf("\tFile [%s] locked by other process!\t[LOCKED]\n", f);
      return 1;
   }

   time_t t = time(NULL);
   tm *info = localtime(&t);
```

```
   char s[100];
   strftime(s, 100, "%A, %B %d, %H:%M", info);
   printf("Date: %s\n\n", s);


   if(lockf(fd, F_LOCK, 0) == -1) printf("Critical error\n");
   printf("Locking [%s] for 30 seconds\n\n", f);
   for(int i = 1; i <= 30; ++i) {
      printf("□");
      fflush(stdout);
      sleep(1);
   }

   lockf(fd, F_ULOCK,0);
   printf("\n\n\tUnlocked [%s]\n", f);

   return 0;
}
```

Ejercicio 16 **(Opcional).** `flock(1)` proporciona funcionalidad de cerrojos antiguos BSD en guiones *shell*. Consultar la página de manual y el funcionamiento del comando.

Directorios

Ejercicio 17. Escribir un programa que cumpla las siguientes especificaciones:

- El programa tiene un único argumento que es la ruta a un directorio. El programa debe comprobar la corrección del argumento.
- El programa recorrerá las entradas del directorio de forma que:
    - Si es un fichero normal, escribirá el nombre.
    - Si es un directorio, escribirá el nombre seguido del carácter `'/'`.
    - Si es un enlace simbólico, escribirá su nombre seguido de `'->'` y el nombre del fichero enlazado. Usar `readlink(2)` y dimensionar adecuadamente el *buffer*.
    - Si el fichero es ejecutable, escribirá el nombre seguido del carácter `'*'`.
- Al final de la lista el programa escribirá el tamaño total que ocupan los ficheros (no directorios) en kilobytes.

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

int main(int argc, char *argv[]){
        if(argc < 2){
        printf("Missing dirname\nUsage: [PATH/dirname]\n");
        return 1;
        }
```

```
        char * f = argv[1];
        printf("\nUsing [%s]\n\n",  f);

        DIR * dir = opendir(f);

        if(dir == NULL){
        printf("\nError reading [%s]\n\n",  f);
        return 1;
        }

        struct dirent * entry = readdir(dir);
        struct stat s;
        size_t path_size = strlen(f);
        unsigned long long int size = 0;

        printf("%s\n", f);
        while(entry != NULL){
        char *path_to_entry = (char *) malloc(sizeof(char)*(path_size + strlen(entry->d_name) +
3));
        strcpy(path_to_entry, f);
        strcat(path_to_entry, "/");
        strcat(path_to_entry, entry->d_name);
        if(stat(path_to_entry, &s) == -1){
        printf("Critical error\n");
        free(path_to_entry); closedir(dir);
        return 1;
        }
        if(S_ISREG(s.st_mode)){
        printf("\t%s*\n", entry->d_name);
        size += s.st_size;
        }
        else if (S_ISDIR(s.st_mode)){
        printf("\t%s/\n", entry->d_name);
        }
        else if (S_ISLNK(s.st_mode)){
        char *lname = (char *)malloc(s.st_size + 1);
        readlink(path_to_entry, lname, s.st_size + 1);
        printf("\t%s->%s\n", entry->d_name, lname);
        free(lname);
        }

        free(path_to_entry);
        entry = readdir(dir);
        }

        printf("\n\t Total size: %llu kilobytes\n", size/1000);
        closedir(dir);

        return 0;
}
```