Ejercicios Módulo 2.1 - Manejo de ficheros

In [ ]:

```
%%bash
rm -rf *.c *.o *.elf prueba Makefile
```

Cuestiones

Cuestión 1 - Creación de ficheros. Ejecuta la orden `touch prueba`. ¿Con qué permisos se ha creado el nuevo fichero?. Ahora ejecuta `umask`, ¿qué máscara se está usando para fijar los permisos de un fichero nuevo?, ¿qué argumentos estará usando `touch` en su llamada a `open()` para que se reproduzca el comportamiento observado?. Modifica la máscara para que `touch` cree ficheros con permisos de lectura y escritura para usuario y grupo, pero sólo de lectura para el resto.

Respuesta:

Permisos: -rw-r--r-- 1 usuario_local users 0 mar 2 11:29 prueba Máscara: 0022 Argumentos de touch: 0644 Modificar Máscara: umask 0113

In [ ]:

```
%%bash
# Comandos que justifican tu respuesta
touch prueba
ls -l
umask
```

Cuestión 2 - Tamaño ficheros. Crea un nuevo fichero usando la orden:

```
    echo "Hola mundo!!"  > nuevoFichero
```

¿Qué tamaño tiene ese fichero?

A continuación usa la orden:

```
    stat -c " %s %b %B %o" nuevoFichero
```

¿Qué significa cada número de los que aparecen? ¿Cuánto ocupa realmente el fichero en disco?

Respuesta:

Tamaño: 4,0K nuevoFichero %s: total size in bytes %b: number of blocks allocated %B: size in bytes of each block reported by %b %o: optimal I/O transfer size hint Realmente el fichero ocupa 8 bloques de 12 bytes

In [ ]:

```
%%bash
# Comandos que justifican tu respuesta
du -sh *
man stat 7
echo "Hola mundo!"  > nuevoFichero
stat -c " %s %b %B %o" nuevoFichero
```

Cuestión 3 - Enlaces. Crea un enlace simbólico (`man ln`) al fichero `nuevoFichero` llamado `miLink`. ¿Cuánto ocupa el fichero del enlace? Crea un nuevo enlace simbólico al mismo fichero pero usando la ruta completa hasta `nuevoFichero`. ¿Cuánto ocupa éste? Explica la diferencia. ¿Coincide alguno de los tamaños con el del fichero original?

Respuesta:

El fichero del enlace ocupa 12 bytes El nuevo enlace simbólico ocupa 25 bytes La diferencia es por guardar la ruta entera NuevoFichero ocupa 8 bloques de 12 bytes, coincide con el primer enlace simbolico en el numero de bytes por bloque

In [ ]:

```
%%bash
# Comandos que justifican tu respuesta:
ln -s nuevoFichero milink
stat milink
ln -s /home/hlocal/nuevoFichero milink2
stat milink2
stat nuevoFichero
```

Cuestión 4 - Nodos-i. Utiliza `ls -li` para mostrar la información de los ficheros del directorio actual, incluyendo su número de nodo-i. Los enlaces simbólicos creados anteriormente, ¿tienen el mismo número de nodo-i que el fichero al que apuntan? Crea ahora un enlace rígido al mismo fichero `nuevoFichero`. ¿Coincide su número de nodo-i con el de alguno de los anteriores?

Respuesta:

En los enlaces simbolicos varia el inodo respecto al nuevoFichero. El enlace rigido si mantiene el mismo inodo que el fichero fuente ya que es exactamente igual

In [ ]:

```
%%bash
# Comandos que justifican tu respuesta:
```

Cuestión 5 - Redirección. Considera nuevamente la orden

```
echo "Hola mundo!!"  > nuevoFichero
```

¿Qué efecto tiene el símbolo >? ¿Qué código estará ejecutando `bash` para conseguir el comportamiento observado? Investiga cómo redireccionar la salida de error estándar en `bash`.

Respuesta: comando 2 > error.log

Este primer método SIEMPRE crea el fichero error.log, pero solo tendrá contenido si la ejecución del comando fallase por cualquier motivo.

La salida por consola es suprimida y para ver el motivo del fallo deberíamos inspeccionar el fichero error.log.

---

para redireccionar > se ejecutara un dup con el cambio de descriptor.

In [ ]:

```
%%bash
# Comandos que justifican tu respuesta
```

Apertura y creación de ficheros

Ejercicio 1 - open.c Escribe un programa que simplemente abra un fichero existente utilizando la función `open`. Varía los `flags` de apertura, y a continuación intenta responder a las siguientes preguntas (consulta la página de manual para responder a alguna de las preguntas):

1. ¿Qué combinación de flags para la función `open` serían equivalentes a la invocación de la función `creat`?
2. ¿Qué flag o combinación de flags serían necesarios para que la función `open` devolviese un error si se intenta crear un fichero que ya existe?
3. ¿Bajo qué circunstancia la función `open` devolverá un error de tipo `ENOENT`? En esa circunstancia, ¿qué flag evitaría dicho error?
4. ¿Bajo qué circunstancia o circunstancias (combinación de `flags` y tipo de fichero), la función `open` devolverá un error de tipo `EISDIR`?
5. ¿Bajo qué circunstancia la función `open` devolverá un error de tipo `ENAMETOOLONG`?

Respuesta:

In [ ]:

```
%%writefile open.c
/*****************************************************************************/
// includes:

/*****************************************************************************/

int main() {
/*
Definicion de los descriptores. Utiliza tantos descriptores y archivos como
necesites para responder a las preguntas. Se pueden reutilizar descriptores.
*/
  int  fd1, fd2;

/*****************************************************************************/
// Código para responder a la pregunta 1:
   fd1 = open("prueba", O_WRONLY|O_CREAT|O_TRUNC);

/*****************************************************************************/

/*****************************************************************************/
// Código para responder a la pregunta 2:
   fd1 = open("prueba", O_WRONLY|O_CREAT|O_EXCL);
   if (fd1<0){
    if(errno==EEXIST){
         perror( "open EEXIST" );
    }
   }

/*****************************************************************************/

/*****************************************************************************/
// Código para responder a la pregunta 3:
 fd1 = open("prueba", O_WRONLY);
  if (fd1<0)         /* error */ //-1 si errno==ENOENT fichero no existe;
    if(errno==ENOENT){
        perror( "open ENOENT" );
    }

/*****************************************************************************/

/*****************************************************************************/
// Código para responder a la pregunta 4:
  fd1 = open("/home", __O_DIRECTORY | O_RDWR);
  if (fd1<0)         /* error */ //-1 si errno==ENOENT fichero no existe;
    if(errno==EISDIR){
        perror( "open EISDIR" );
    }
  }

/*****************************************************************************/

/*****************************************************************************/
// Código para responder a la pregunta 5:
    char d_name[NAME_MAX + 5];
    fd1 = open(d_name,O_WRONLY);
    if (fd1<0){
      if(errno==ENAMETOOLONG){
        perror( "open ENAMETOOLONG" );
      }
    }

/*****************************************************************************/
close(fd1);
/*****************************************************************************/
// Cierre de los ficheros:

/*****************************************************************************/

  return 1;
}
```

```
}
```

In [ ]:

```
%%bash
gcc open.c -o open.elf
./open.elf
```

Descriptores de ficheros

Ejercicio 2 - dup.c Completa el programa para que escriba en un fichero la frase `Hola, mundo`. No está permitido utilizar directamente la llamada al sistema `write` ni la función `fwrite()` (se debe mantener la llamada a `printf()` tal y como está). Propón dos alternativas distintas para resolver el problema.

Respuesta:

In [ ]:

```
%%writefile dup.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {

 /* Descriptor del fichero */
  int fd;

  // apertura del fichero de salida
  fd = open("prueba", O_CREAT|O_WRONLY|O_TRUNC, 0660);

  if( fd == -1 ) {
    perror("open");
    exit(1);
  }

 /* Manipular los descriptores de ficheros para que
 la llamada a printf del final se escriba en el fichero  */
  if(dup2(fd,STDOUT_FILENO) == -1){
  perror("fallo dup2");
   exit(EXIT_FAILURE);

  }
 /* Cerramos el fichero, puesto que no podemos usar su
  descriptor de acuerdo al enunciado*/
  close(fd);

/* Prueba */
  printf("Hola , mundo\n");
}
```

In [ ]:

```
%%bash
gcc dup.c -o dup.elf
./dup.elf
```

Ejercicio 3 - cat.c Escribe una función llamada `copy`, que no reciba ningún parámetro. Dicha función leerá continuamente un carácter desde el descriptor de fichero `0`, y escribirá en el descriptor de fichero `1` cada cadena de caracteres introducida. Invocar dicha función desde `main()` para mostrar por pantalla el contenido de un determinado fichero.

Respuesta:

In [9]:

```
%%writefile cat.c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>


// funcion que va leyendo
// de la entrada estandar byte a byte
// y escribiendo lo leido en la salida
// estandar (hasta que no se detecte nada
// en la entrada estandar
void copy() {
    char c;
    int error = 0;

    do{
       if((read(0, &c, 1) != 1) || (write(1, &c, 1) != 1))     error = 1;
    }  while(c != '\0' && !error);

}


int main( int argc, char **argv)
{
  int    fd;
  char ch;

/* Abrir el fichero que se quiere mostrar*/
 fd = open("prueba", O_RDONLY);
/* Hacer lo necesario para que, al llamar
 a copy, se muestre su contenido por pantalla*/
  dup2(fd, 0);

  copy();
  return 0;
}
```

```
Overwriting cat.c
```

In [10]:

```
%%bash
gcc cat.c -o cat.elf
echo "Hola mundo!" > prueba
./cat.elf
```

```
Hola mundo!
```

Desplazamiento en ficheros

Ejercicio 4 - lseek.c Ejecuta el fichero que crea un agujero (*hole*) de un determinado tamaño en un fichero. Abre el fichero con un editor de textos y comprueba el resultado. Utiliza el comando `hexdump` para mostrar el contenido del fichero.

Respuesta:

Salida tras hexdump -c prueba: Principio del archivo 0 Posicion despues de escribir 5 Otra vez al principio 0 Escribimos a partir de 16384 0000000 H O L A , \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 0000010 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 * 0004000 m u n d o 0004006

In [3]:

```
%%writefile lseek.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
 off_t inicio;
    char linea1[10]="HOLA,";
    char linea2[10]=" mundo";
    int fd;

    fd=open("prueba",O_WRONLY|O_CREAT|O_TRUNC,0666);

    inicio = lseek(fd,0,SEEK_CUR);
    printf("Principio del archivo %ld\n", inicio);


    write(fd, linea1, 5);

    printf("Posicion despues de escribir %ld\n",lseek(fd,0,SEEK_CUR));

    lseek(fd,inicio,SEEK_SET);
    printf("Otra vez al principio %ld\n",lseek(fd,0,SEEK_CUR));

    // saltamos 16KB
    lseek(fd,16384,SEEK_SET);

    printf("Escribimos a partir de %ld\n",lseek(fd,0,SEEK_CUR));
    // vuelvo a escribir
    write(fd, linea2, 6);
    close(fd);

    return 1;
}
```

```
Writing lseek.c
```

In [8]:

```
%%bash
gcc lseek.c -o lseek.elf
./lseek.elf
# Comandos que justifican tu respuesta:
hexdump -c prueba
```

```
Principio del archivo 0
Posicion despues de escribir 5
Otra vez al principio 0
Escribimos a partir de 16384
```

Lectura/escritura en ficheros

Ejercicio 5 - rw.c Escribe un programa que copie un fichero usando las llamadas al sistema `read` y `write`. Usa `time` para temporizar y ejecuta varias veces el código con el mismo fichero origen aumentando el número de bytes leídos/escritos en cada llamada al sistema. Para ello usa una macro `TAM_BLOQUE` cuyo valor se especificará en tiempo de compilación usando el flag `-D` (prueba con valores de `TAM_BLOQUE` de 1, 64, 512, 1024 y 4096, usando como entrada un fichero de varios MB (por ejemplo, copia el fichero `/boot/initrd.img-3.2.0-4-amd64` a tu directorio de trabajo y usa esa copia para probar). ¿Varía significativamente el tiempo de copia?

Respuesta:

In [3]:

```
%%writefile rw.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>


#ifndef TAM_BLOQUE
#define TAM_BLOQUE 1
#endif

// declarar array para almacenar los bytes leidos
char buffer[];

int main() {

  int   fdo, fdd,n;
  char  c;

   // mantener este fichero para pruebas iniciales.
  // cambiar por uno mayor para medir tiempos
  fdo = open("/etc/passwd", O_RDONLY);

  if( fdo == -1 ) {
    perror("open");
    exit(1);
  }

  // Abrir fichero destino para escritura
  fdd = open (argv[2],O_WRONLY|O_CREAT|O_TRUNC);
  if (fdd == -1) {
        perror("Error al abrir el fichero de destino");
  }

  printf("Comienza la copia con TAM_BLOQUE=%d bytes\n", TAM_BLOQUE);
  // Incluir las llamadas de temporizacion de vuestra libreria (start...)

  // Bucle de copia: leer TAM_BLOQUE btyes por llamada. Escribir lo leido
  while ((n=read(fdo,buffer, TAM_BLOQUE)) > 0 )
  {
    if (write(fdd, buffer, n) != n)
     {
      perror("Error en write");
  }
  }

  if (n == -1)
    perror("read");

  close(fdd);
    close(fdo);
}
```

```
Overwriting rw.c
```

In [2]:

```
%%bash
# Lista de comandos que justifican tu respuesta:
dd if= /dev/zero of=./prueba bs=512 count=$((16*1024*1024/512))
rm -f prueba2
gcc  rw.c -o rw.elf -DTAM_BLOQUE=1
time ./rw.elf prueba prueba2
```

Enlaces

Ejercicio 6 - lee_enlace.c Escribe un programa con nombre `lee_enlace.c` que, para un determinado enlace simbólico, muestre por pantalla el nombre del fichero al que apunta dicho enlace. ¿Qué se almacena en el campo `st_size` del inodo correspondiente a dicho fichero? Ten especial precaución en la reserva de espacio de memoria para la cadena que contendrá el nombre del fichero destino.

Respuesta:

In [ ]:

```
%%writefile lee_enlace.c
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdlib.h>

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

  char *buf;
  struct stat statbuf;
  int n;

/* leemos las propiedades del link mediante lstat */
  if (lstat(argv[1], &statbuf) == -1) {
    perror("lstat");
    exit(1);
  }
/* Comprobamos si es un link o no */
  if (!S_ISLNK(statbuf.st_mode))  {
    fprintf(stderr, "%s is not a symbolic link.\n",
            argv[1]);
  exit(1);
  }

/* Reservamos espacio para el link (+1 final de cadena */
  buf = (char *)malloc(statbuf.st_size + 1);

  if (buf == NULL) {
    fprintf(stderr, "Out of memory.\n");
    exit(1);
  }

/* Leemos el link */
  n = readlink(argv[1], buf, statbuf.st_size + 1);

  if (n == -1) {
    perror("readlink");
    exit(1);
  }

/* Imprimimos el link buffer */
  buf[n] ='\0' //Caracter de fin de cadena
  printf("%s\n", buf);
  exit(0);

}
```

In [ ]:

```
%%bash
gcc lee_enlace.c -o lee_enlace.elf
# Lista de comandos que justifican tu respuesta:
rm -f fichero enlace
echo "Hola" > fichero
ls -s fichero enlace
./lee_enlace.elf enlace
./lee_enlace.elf fichero
```

Ejercicio 7 - links.c Implementa un programa que cree un fichero, y a continuación un enlace simbólico y otro duro que apunten a él. Extrae el tamaño de cada uno de dichos ficheros. ¿Son diferentes en ambos casos? ¿Por qué?

Respuesta: Si son distintos, porque el enlace duro se añade al i-node del archivo por lo que no crea otro bloque donde guardarse, por otro lado el enlace simnolico se guarda en otra direccion de memoria distinta al propio archivo. que apunta.

In [ ]:

```
%%writefile links.c
```

```
#include <unistd.h>

int main() {
  int  fd1;

  char *buf;
  struct stat statbuf;
  int n;
  //Crea un fichero (open)
  fd1 = open("prueba", O_WRONLY|O_CREAT|O_TRUNC);

  /* Creacion de un enlace hard */
  link("prueba", "pruebaLinkH");

  /* Enlace simbolico */
  symlink("prueba", "pruebaLinkS");

   lstat("pruebaLinkH",&statbuf);
   printf("Tamaño total en bytes pruebaLinkH:%ld \n",statbuf.st_size);
   printf("Tamaño total en bloques pruebaLinkH:%ld \n",statbuf.st_blocks);


   lstat("pruebaLinkS",&statbuf);
   printf("Tamaño total en bytes pruebaLinkS:%ld \n",statbuf.st_size);
   printf("Tamaño total en bloques pruebaLinkS:%ld \n",statbuf.st_blocks);
  return 0;
}
```

In [ ]:

```
%%bash
gcc links.c -o links.elf
# Lista de comandos que justifican tu respuesta:
```

Manejo de directorios

Ejercicio 8 - mi_ls.c Escribe un programa llamado `mi_ls.c` que, para un determinado directorio, muestre por pantalla su contenido (nombres de los ficheros que lo componen).

In [ ]:

```
%%writefile mi_ls.c
#include <stdio.h>
#include <sys/stat.h>
#include <dirent.h>

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

  struct stat buf;
  DIR   *dirp;
  struct dirent *dent;
  int n;

/* obtencion de la descripcion del directorio */
  if (stat(argv[1], &buf) == -1) {
    perror("stat");
    exit(1);
  }

/* Comprobamos que es un directorio */
  if (!S_ISDIR(buf.st_mode)) {
    fprintf(stderr, "%s no es un directorio.\n", argv[1]);
    exit(1);
  }

/* Apertura del directorio usando operdir*/
  if ((dirp = opendir(argv[1])) == NULL) {
    perror("opendir");
    exit(1);
  }
  printf("<<<< Contenidos de %s>>>>\n",argv[1]);

/* Leemos el directorio */
```

```
  while ((dent = readdir(dirp)) != NULL) {
    printf("%d\t%d\t%d\t%s\n", dent->d_ino, dent->d_off, dent->d_reclen, dent->d_name);
  }

  // Cerrar directorio
  closedir(dirp);
}
```

In [ ]:

```
%%bash
gcc mi_ls.c -o mi_ls.elf
./mi_ls.elf
```

Ejercicio 9 - mi_ls_bis.c A partir del programa anterior, crea un nuevo programa que, además, muestre por pantalla la información de los permisos de cada fichero tal y como la muestra el comando `ls -l`.

In [ ]:

```
%%writefile mi_ls_bis.c
/*****************************************************************************/

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

void imprime_permisos(struct stat estru){
 printf((S_ISDIR(estru.st_mode))?  "d" : "-");
 printf((estru.st_mode & S_IRUSR)? "r" :  "-");
 printf((estru.st_mode & S_IWUSR)? "w" : "-");
 printf((estru.st_mode & S_IXUSR)? "x" : "-");
 printf((estru.st_mode & S_IRGRP)? "r" : "-");
 printf((estru.st_mode & S_IWGRP)? "w" : "-");
 printf((estru.st_mode & S_IXGRP)? "x" : "-");
 printf((estru.st_mode & S_IROTH)? "r" : "-");
 printf((estru.st_mode & S_IWOTH)? "w" : "-");
 printf((estru.st_mode & S_IXOTH)? "x" : "-");
}

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

  struct stat buf;
  DIR   *dirp;
  struct dirent *dent;
  int n;

/* obtencion de la descripcion del directorio */
  if (stat(argv[1], &buf) == -1) {
    perror("stat");
    exit(1);
  }

/* Comprobamos que es un directorio */
  if (!S_ISDIR(buf.st_mode)) {
    fprintf(stderr, "%s no es un directorio.\n", argv[1]);
    exit(1);
  }

/* Apertura del directorio usando operdir*/
  if ((dirp = opendir(argv[1])) == NULL) {
    perror("opendir");
    exit(1);
  }
  printf("<<<< Contenidos de %s>>>>\n",argv[1]);

/* Leemos el directorio */
  while ((dent = readdir(dirp)) != NULL) {
    printf("%d\t%d\t%d\t%s\n", dent->d_ino, dent->d_off, dent->d_reclen, dent->d_name);
     stat(dent->d_name,&buf);
    if(S_ISDIR(buf.st_mode)){
        imprime_permisos(buf);
    }
```

```
  }

  // Cerrar directorio
  closedir(dirp);
}
/*****************************************************************************/
```

In [ ]:

```
%%bash
gcc mi_ls_bis.c -o mi_ls_bis.elf
./mi_ls_bis.elf
```

Makefile

Ejercicio 10 - Makefile. Crea un fichero **Makefile** que compile todos los ejemplos del directorio de modo que se generen ejecutables con el mismo nombre que el fuente, pero con extensión **.elf**. El **Makefile** deberá incluir una regla **clean** que borre todos los ficheros **.o** y **.elf** generados.

In [ ]:

```
%%writefile Makefile
CC = gcc
CFLAGS = -g -Wall -Werror

#################################################

all: mi_ls_bis.elf  mi_ls.elf links.elf lee_enlace.elf cat.elf lseek.elf
    dup.elf open.elf rw.elf

%.elf: %.o
 $(CC) $(CFLAGS) $^ -o $@

&.o: %.c
 $(CC) -c $< -o $@

.PHONY: clean


clean:
 rm -f *.o *.elf
#################################################
```

In [ ]:

```
%%bash
make clean
make
```