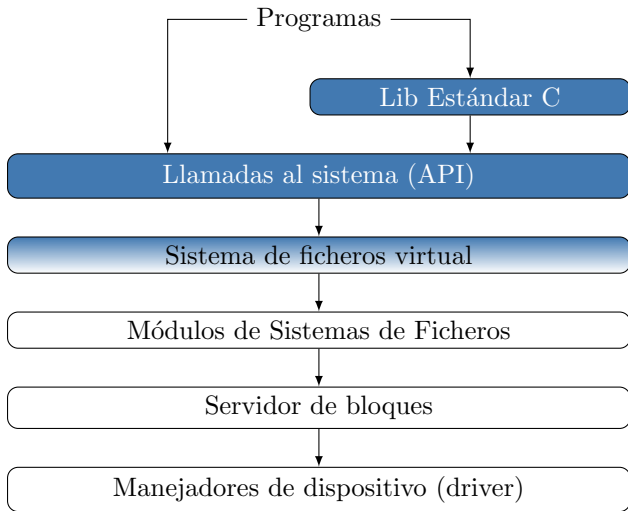


# Sistemas Operativos



Gestion de Ficheros  
Ficheros, Directorio y sus APIs

# Gestión de Ficheros



# Agenda



- 1** Ficheros
- 2** Biblioteca Estándar de C
- 3** Directorios
- 4** Enlaces duros y simbólicos
- 5** Utilidades Linux/UNIX de interés
- 6** FAQ

# Agenda



- 1 Ficheros**
- 2 Biblioteca Estándar de C
- 3 Directorios
- 4 Enlaces duros y simbólicos
- 5 Utilidades Linux/UNIX de interés
- 6 FAQ

# Concepto de Fichero



- Unidad lógica de almacenamiento no volátil
  - Datos relacionados entre sí, reconocidos como una única cosa
  - Identificado por su nombre o ruta
- Suelen utilizarse como:
  - fuente de entrada de datos a los programas
  - almacenamiento a largo plazo de las salidas de los programas
- POSIX utiliza la abstracción de ficheros también para:
  - Comunicar procesos (pipes y sockets)
  - Interactuar con drivers de dispositivos (ficheros de dispositivos)

# Tipos de ficheros en POSIX



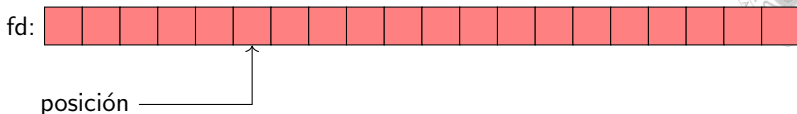
- Regular
- Directorio
- Enlace Simbólico
- Tubería (Pipe)
- Fichero de dispositivo (caracteres o bloques)
- Socket

# Atributos de un fichero en POSIX



- Nombre (nombre + extensión)
- Tipo
- Tamaño
- Fecha de creación
- Fecha de última modificación
- Propietario, grupo
- Permisos de lectura, escritura y ejecución
  - distintos para propietario, grupo y otros

# Modelo



El modelo de fichero que el SO presenta al programador es:

- una secuencia ordenada de bytes
  - byte **n**: saltar **n** bytes desde el comienzo
- un marcador o puntero de posición en dicha secuencia
  - no confundir con una variable tipo puntero
  - una lectura o escritura hacen avanzar este puntero
- un descriptor de fichero (fd) para manejarlo
  - Asociado a un apertura concreta del fichero (permisos y posición)
- unas operaciones básicas sobre el fichero



# POSIX: descriptor de fichero



En POSIX un fichero se maneja con un descriptor de fichero obtenido al abrirlo con la siguiente llamada al sistema:

```
int open(const char *path, int oflag, ...);
```

El descriptor es un entero:

- Indica la posición correspondiente en la Tabla de Descriptores de Ficheros Abiertos (TFA) del proceso
- La TFA es una estructura gestionada por el SO para el proceso
- La apertura se realiza en un modo indicado por los flags
  - Máscara de bits
  - Permisos de lectura y/o escritura
  - Qué hacer si existe (colocarse al final, borrar, ...) o si no existe

Se recomienda consultar la página de manual de `open` (`# man 2 open`)



# POSIX: operaciones sobre ficheros

- **open, creat**: apertura y/o creación
- **close**: cierre
- **lseek**: desplazamiento del puntero de posición
- **read**: lectura desde la posición actual
- **write**: escritura desde la posición actual
- **unlink**: borrado de archivo (nombre)
- **stat**: obtención de atributos de un fichero

Se recomienda consultar las páginas de manual de estas llamadas al sistema



## Ejemplo: copia de ficheros

```
int copy(char* src, char* dst){
    int fd1, fd2, n;
    unsigned char c;
    if ((fd1 = open(src, O_RDONLY)) == -1){
        perror("Error al abrir src");
        return -1;
    }
    if ((fd2 = open(dst, O_WRONLY | O_CREAT), 0660) == -1){
        perror("Error al abrir dst");
        close(fd1);
        return -1;
    }
    do {
        n = read(fd1, &c, 1);
        if (n > 0)
            n = write(fd2, &c, 1);
    } while (n > 0);
    close(fd1);
    close(fd2);
    if (n < 0){
        perror("Error en la copia");
        return -1;
    }
    return 0;
}
```

# Agenda



- 1 Ficheros
- 2 Biblioteca Estándar de C
- 3 Directorios
- 4 Enlaces duros y simbólicos
- 5 Utilidades Linux/UNIX de interés
- 6 FAQ

# Biblioteca Estándar de C (Stdlib)



Ofrece servicios independientes de sistema:

- Operaciones básicas sobre ficheros
  - `fopen`, `fclose`, `fread`, `fwrite`, `fflush`, ...
- Operaciones de entrada/salida estándar:
  - `printf`, `sprintf`, `snprintf`, `fprintf`, `scanf`, `fscanf`, ...
  - `getc`, `fgetc`, `fgets`, `getchar`, `ungetc`, ...
- Operaciones de procesamiento de cadenas de caracteres
  - `strlen`, `strcat`, `strcmp`, `strcpy`, ...
- Operaciones de gestión de memoria:
  - `malloc`, `realloc`, `calloc`, `free`, `memcpy`, `memset`, ...
- Operaciones matemáticas
  - `sin`, `cos`, `tan`, `sqrt`, `rand`, ...



## Ejemplo: fopen vs. open

```
int open(const char *pathname, int flags);
```

- Se usa un entero como identificador de la apertura
  - Es una entrada de una tabla de descriptores abiertos del proceso
- Sólo válida en sistemas POSIX
- El argumento flags vinculado con las opciones de apertura de los sistemas POSIX

```
FILE *fopen(const char *pathname, const char *mode);
```

- Se usa una estructura FILE como identificador de la apertura
  - fopen devuelve la dirección (puntero) de la estructura usada
- Válido en todos los sistemas.



# Buffer intermedio

Las llamadas al sistema son costosas:

- Cada llamada a `open`, `close`, `read`, `write`, ... supone una excepción software

La biblioteca estándar de C usa buffers intermedios

- Llamada consecutivas a `fread` o `fwrite` suponen lecturas o escrituras del buffer intermedio.
- Cuando se cumple alguna condición se hace la llamada al sistema para leer o escribir del fichero real (escritura/lectura en bloque):
  - Porque se llena o vacía el buffer
  - Porque escribimos un final de línea

La política se puede cambiar con `setvbuf`

```
int setvbuf(FILE *stream, char *buf,  
            int mode, size_t size);
```

# Versiones seguras con n



Ejemplo: strcat y strncat

```
#include <string.h>
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

Ambas funciones concatenan dos cadenas de caracteres, copiando la cadena apuntada por src a continuación de dest.

- strcat: copia los bytes hasta encontrar el fin de línea de src.
- strncat: copia como **máximo n** bytes.

Si src es un dato de entrada, nunca podemos estar seguros de cuál será su tamaño, debemos limitar la escritura al tamaño del buffer apuntado por dest





# Escritura en ascii vs binaria

Si tenemos el siguiente programa:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int a = 5;
    fwrite(&a, sizeof(a), 1, stdout);
    return 0;
}
```

y lo ejecutamos desde un terminal...

- ¿se verá algo en el terminal?
- ¿y si volcamos la salida en un fichero y lo abrimos?
- ¿y si le hacemos un cat?

# Escritura en ascii vs binaria



Escritura binaria

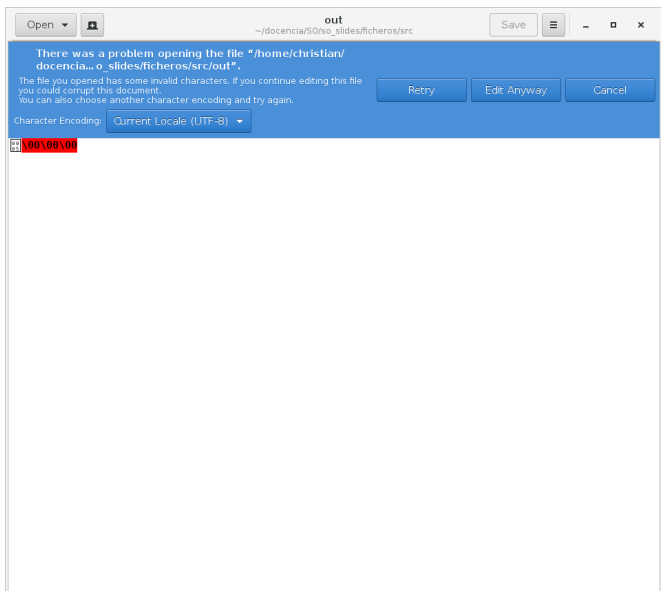
```
$ ./binwrite
$ ./binwrite | tee
$ ./binwrite > out
$ cat out
$ hexdump -C out
00000000  05 00 00 00
00000004
```

|....|

# Escritura en ascii vs binaria



Si lo editamos con un editor:





## Escritura en ascii vs binaria

Repitamos el ejercicio con el siguiente programa:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a = 5;
    fprintf(stdout, "%d", a);
    return 0;
}
```

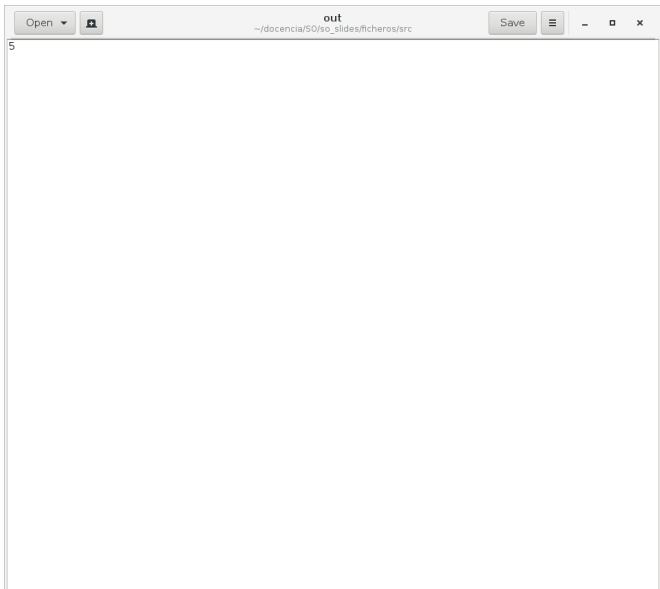
¿Qué pasa ahora?

Escritura ascii

```
$ ./asciwrite
5$ ./asciwrite | tee
5$ ./asciwrite > out
$ cat out
5$
```

# Escritura en ascii vs binaria

Si lo editamos con un editor:



## Ejemplo: lectura de un string



- Queremos una función que reciba un `stream` del que debe leer una cadena de caracteres C válida, y escribirla en un buffer de salida, cuya dirección y tamaño se pasan como argumento.
- La función devolverá el número de bytes escritos en el buffer. Si la cadena es más larga que el tamaño del buffer interrumpirá la lectura cuando se llene dicho buffer.
- La cadena devuelta será siempre una cadena C bien formada (terminada en carácter null).

## Ejemplo: lectura de un string



```
int find_str(char *buf, size_t size, FILE *stream)
{
    int bread, n;

    if (size <= 0)
        return 0;

    bread = 0;
    do {
        if ((n = fread(buf, 1, 1, stream)) == 1){
            bread++;
            buff++;
        }
    } while ((bread < size) && (n > 0) &&
        (*(buf-1) != '\0'));

    *(buf-1) = '\0';
    return bread;
}
```

# Agenda



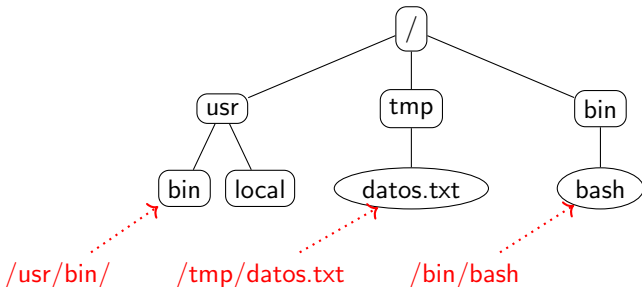
- 1 Ficheros
- 2 Biblioteca Estándar de C
- 3 Directorios**
- 4 Enlaces duros y simbólicos
- 5 Utilidades Linux/UNIX de interés
- 6 FAQ





# Concepto de Directorio

- Fichero especial para agrupar ficheros y directorios relacionados
- Es una relación lógica
  - El directorio contiene una *lista/relación* de nombres de ficheros, no *contiene* los ficheros en sí
  - Los ficheros tampoco se almacenan físicamente juntos en el disco
- El conjunto de directorios del SF forma una estructura de árbol
- El nombre completo o ruta de un fichero se forma con la lista de directorios que hay que atravesar para llegar al fichero desde la raíz.



# Modelo



El modelo de directorio que el SO presenta al programador es:

- Una tabla de entradas de directorio, donde cada entrada contiene:
  - nombre de un fichero/directorio
  - información adicional dependiente del SF (info)
- Un descriptor de directorio
- Una serie de operaciones específicas para directorios
  - Ejemplo: `readdir` nos da la siguiente entrada de directorio

nombre1	info
nombre2	info
nombre3	info
nombre4	info
nombre5	info
nombre6	info
nombre7	info

# Entradas especiales



Todos los sistemas los directorios suelen tener dos entradas especiales:

- `.` el propio directorio
- `..` el directorio padre

Estas entradas permiten:

- Nombrado de ficheros y directorios con rutas relativas
  - `/tmp/datos.txt`
  - `./datos.txt`
  - `../tmp/datos.txt`
- Recorrido del árbol de directorios

# POSIX: descriptor de directorio



En POSIX un directorio se maneja con el descriptor obtenido al abrirlo:

```
DIR* opendir(char *dirname);
```

- El descriptor es una estructura DIR
- Su almacenamiento en memoria lo gestiona la biblioteca del sistema

Se recomienda consultar la página de manual de `opendir`  
(# `man opendir`)

# POSIX: abstracción de entrada de directorio



```
struct dirent* readdir(DIR* dirp);
```

La función `readdir` devuelve la siguiente entrada de directorio como una estructura `struct dirent`

- Almacenamiento gestionado por la biblioteca del sistema
- Implementación dependiente del sistema
- POSIX fija que debe tener al menos dos campos:
  - `d_name`: nombre del fichero/directorio
  - `d_ino`: nodo-i del fichero
- Definida en el fichero `dirent.h`

# POSIX: operaciones sobre directorios



- **mkdir**: crea un directorio con un nombre y protección
- **rmdir**: borra el directorio vacío con un nombre
- **opendir**: abre un directorio y se sitúa en la primera entrada
- **closedir**: cierra un directorio
- **readdir**: lee la siguiente entrada del directorio
- **rewinddir**: sitúa el puntero de posición en la primera entrada
- **link/symlink**: crea una nueva entrada en el directorio para un enlace físico o lógico
- **unlink**: elimina una entrada del directorio
- **chdir**: cambia el directorio actual
- **getcwd**: obtener el directorio actual
- **rename**: cambiar el nombre de una entrada del directorio

Se recomienda consultar las páginas de manual de estas funciones

## Ejemplo: búsqueda de fichero en ./



```
int busca(char* name)
{
    struct dirent *dp;
    DIR* dirp = opendir(".");

    if (dirp == NULL)
        return ERROR;

    while ((dp = readdir(dirp)) != NULL) {
        if (strcmp(dp->d_name, name) == 0) {
            closedir(dirp);
            return FOUND;
        }
    }
    closedir(dirp);
    return NOT_FOUND;
}
```

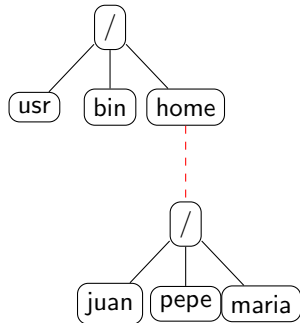
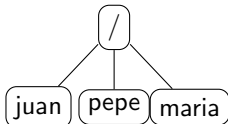
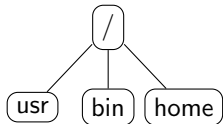
# Árbol único vs Árbol por dispositivo



- Cada dispositivo lógico (volumen, partición,...) tiene un sistema de ficheros independiente
- VMS, Windows
  - Un árbol de directorios independiente por SF
  - La ruta completa de un fichero comienza con un identificador del volumen (Ej: C:\)
- UNIX
  - Se maneja un único árbol de directorios
  - Los directorios raíz de los SF de los volúmenes se *montan* en algún directorio de dicho árbol
  - Ventaja: imagen única del sistema, oculta el tipo de dispositivo
  - Desventaja; dificulta un poco el recorrido del árbol, en cada directorio hay que comprobar si es un punto de montaje o no



## Ejemplo montado de SF



# Agenda



- 1 Ficheros
- 2 Biblioteca Estándar de C
- 3 Directorios
- 4 Enlaces duros y simbólicos**
- 5 Utilidades Linux/UNIX de interés
- 6 FAQ

# POSIX: Enlace Duro



En POSIX existe el concepto de enlace duro, que no es más que un nombre de un fichero

- Un fichero puede tener más de un nombre/ruta/enlace
- Siempre dentro del mismo SF
  - sin *atravesar* puntos de montaje
- Los enlaces son indistinguibles entre sí
- Para borrar el fichero hay que eliminar todos sus enlaces (`unlink`)
  - ejecutar `rm` sólo sobre uno de los nombres no borra el fichero
- No se pueden hacer enlaces duros a directorios

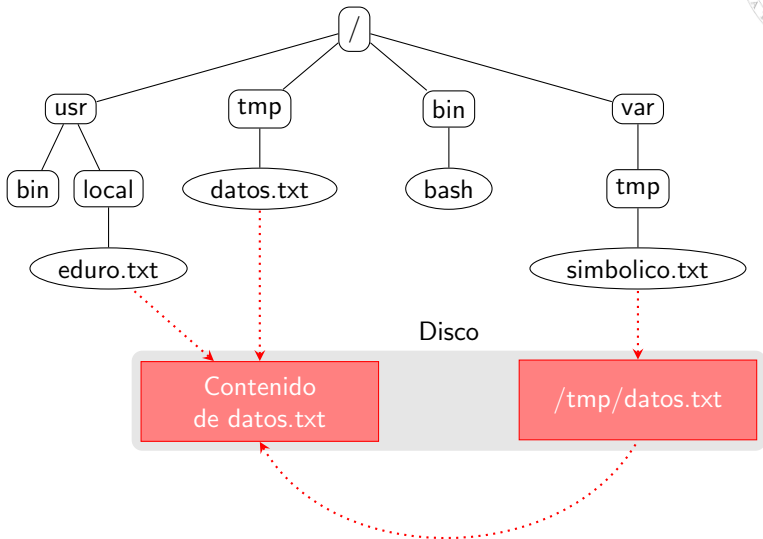
# POSIX: Enlace simbólico



En POSIX existe el concepto de enlace simbólico, que es un tipo especial de fichero cuyo contenido es el nombre del fichero al que apunta

- Un `open` sobre el enlace simbólico hace que el `SO` abra el fichero apuntado, no el enlace
- El nombre escrito en el enlace puede ser absoluto o relativo
- Diferencias con enlace duro:
  - Hay un fichero intermedio
  - Si se borra el fichero apuntado el enlace permanece en el sistema, apuntando a un fichero que ya no existe
    - un `open` sobre el enlace simbólico daría error
  - Si se borra el fichero apuntado y se crea otro fichero con el mismo nombre, el enlace apuntaría al nuevo fichero
  - Borrar el enlace simbólico no afecta al fichero apuntado
  - Podemos atravesar puntos de montaje
  - Podemos hacer enlaces simbólicos a directorios

## Ejemplo: enlace duro vs simbólico



# Agenda



- 1 Ficheros
- 2 Biblioteca Estándar de C
- 3 Directorios
- 4 Enlaces duros y simbólicos
- 5 Utilidades Linux/UNIX de interés**
- 6 FAQ

# Utilidades de interés



- **cp**: copiar ficheros
- **rm**: borrar ficheros
- **cat**: volcar por terminal el contenido ficheros
- **diff**: comparar ficheros y crear un parche con las diferencias
- **tail**: mostrar líneas/bytes del final del fichero
- **head**: mostrar líneas/bytes del comienzo del fichero
- **mkdir**: crear un directorio
- **rmdir**: borrar un directorio vacío
- **cd**: cambiar el directorio de trabajo
- **pwd**: mostrar el directorio de trabajo
- **ln**: crear un enlace duro o simbólico a un fichero
- **mount**: montar un sistema de ficheros en un directorio
- **unmount**: desmontar un sistema de ficheros de su punto de montaje

# Agenda



- 1 Ficheros
- 2 Biblioteca Estándar de C
- 3 Directorios
- 4 Enlaces duros y simbólicos
- 5 Utilidades Linux/UNIX de interés
- 6 **FAQ**



# FAQ



- ¿Qué relación hay entre el modelo lógico y el almacenamiento físico?
- ¿Tiene que estar un fichero almacenado en posiciones contiguas del disco?
- ¿Cómo sabe el SO la posición en el disco de un fichero?
- ¿Qué información adicional necesita para ello? ¿Está en disco o en memoria?
- ¿Cuánto espacio de disco se pierde por la gestión del SF?
- ¿Existen varias alternativas? Y si es así, ¿qué ventajas e inconvenientes tienen?
- ¿Tiene el mismo coste acceder a cualquier posición del fichero?
- ¿Qué supone borrar un fichero? ¿Qué pasa con el contenido que tenía?
- ¿Cómo se implementan los enlaces duros y simbólicos?