Tema 1 – L2: Servicios del Sistema Operativo

1 Introducción

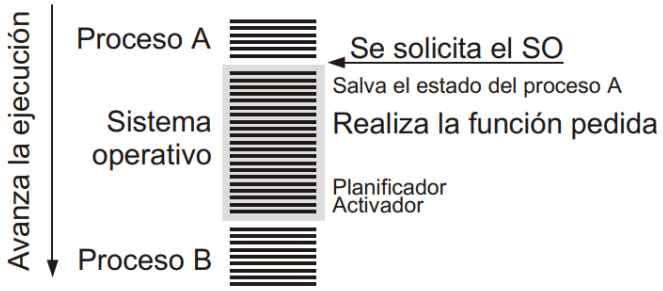
Ejecución del sistema operativo

Una vez finalizado el arranque, el sistema operativo solamente se ejecuta en respuesta a interrupciones. El sistema operativo se activa cuando debe responder a :

- Una petición de servicio de un proceso
- Una interrupción (de un periférico o de reloj)
- Excepciones del hardware

Fases en la activación del Sistema Operativo

1. Se solicita el SO durante un proceso A
2. Se salva el estado del proceso A
3. Se realiza la función pedida
4. Se ejecuta el planificador, que decide que proceso es el siguiente a ejecutar (puede ser el A u otro)
5. Se ejecuta el siguiente proceso



Activación de servicios

Una invocación directa a una rutina del sistema operativo plantea problemas de seguridad. Utilizando una **interrupción software (trap)** se consigue la activación del sistema a operativo en modo seguro. **Rutina de la llamada** de la biblioteca al sistema operativo:

- Se ejecutan las instrucciones máquina que prepara la llamada al SO
- Se ejecuta la instrucción de trap
- Se ejecutan las instrucciones del proceso según los resultados de la llamada al SO

Servicios del SO: Llamadas al sistema

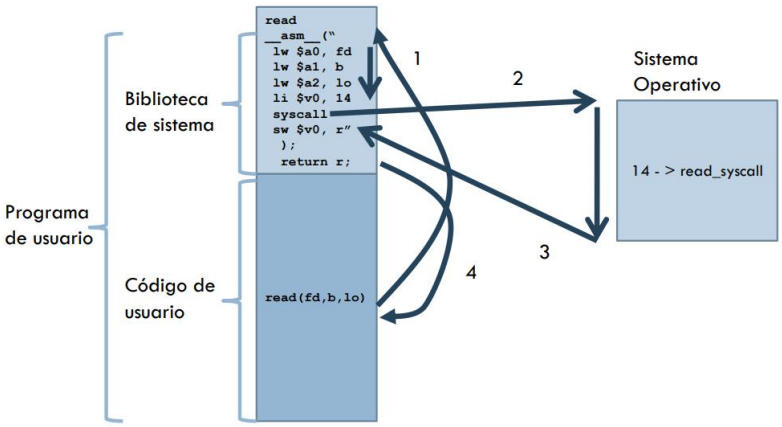
- **Interfaz entre aplicaciones y So.**
  - Generalmente disponibles como funciones en ensamblador
  - Actualmente en otros lenguajes de alto nivel (C, C++, …)
- **Servicios típicos del sistemas operativos**
  - Gestión de procesos
  - Gestión de procesos ligeros
  - Gestión de señales, temporizadores
  - Gestión de memoria
  - Gestión de ficheros y directorios
- **Ejemplos de llamada**
  - read: permite leer datos de un fichero
  - fork: permite crear un nuevo proceso

Invocación de la llamada

- **Cada función** de la interfaz de programación (API) se corresponde con algún **servicio** del sistema operativo
  - La función es un envoltorio para el código que invoca el servicio del sistema operativo
- Incluye la **ejecución** de una **instrucción de trap** que transfiere el control al sistema operativo mediante la generación de la interrupción
- El sistema operativo **trata la interrupción** y devuelve el **control** al programa de **usuario**

Proceso

1. **Se ejecuta la instrucción** a alto nivel que llama a un código almacenado en la biblioteca del sistema
2. Se **ejecuta el conjunto de microinstrucciones en ensamblador**, y se realiza la llamada al sistema operativo que realiza la funcion de leer (ya que el programa y el modo usuario no puede hacerlo)



3. Una vez **se completa la instrucción** en el sistema operativo en modo kernel, se cambia el **modo a usuario** y se **vuelve** a la ejecución de la función de la biblioteca del sistema
4. **Se vuelve al código de usuario** para continuar con el programa cuando se termina el código de la biblioteca del sistema, devolviendo lo que corresponda al usuario

Selección de servicio

Al existir una **única instrucción** de trap y múltiples servicios se hace necesario establecer algún **mecanismo** de **paso de parámetros** entre el proceso de usuario y el núcleo. Como **mínimo** siempre se debe pasar una **especificación del servicio** que se desea ejecutar (típicamente un identificador numérico)

Paso de parámetros

Existen **tres métodos genéricos** para parámetros a las llamadas al sistema:

- En registros
- En una tabla de memoria, cuya dirección se pasa al SO en un registro
- Poner los parámetros en una pila del programa y dejar que el SO los extraiga

Cada sistema proporciona sus propias llamadas al sistema:

- Estándar POSIX en UNIX y LINUX
- Win32 en Windows NT

Rutina de tratamiento

La rutina de tratamiento debe:

- **Recuperar los parámetros** enviados por el proceso de usuario
- **Identificar el servicio** que se desea ejecutar
- **Determinar la dirección** de la rutina de servicio adecuada (indexación en una tabla de rutinas de servicio)
- **Transferir el control** a la rutina de servicio ejecutada por el sistema operativo

Invocación de una llamada (ejemplo)

```
int read (int fd, char*b, int lon){
  int r,
    __asm__(“
      lw $a0, fd
      lw $a1, b
      lw $a2, lon
      li $v0, 14        → PREPARACIÓN DEL TRAP (llamada de lectura)
      syscall           → TRAP
      sw $v0, r”
    );
    return r;}
```

Interfaz del programador

Una interfaz ofrece la **visión** que tiene el **usuario** del sistema operativo. Cada sistema operativo puede ofrecer una o varias interfaces: Linux → POSIX, Windows → Win32 / POSIX

Estándar POSIX

- **Interfaz estándar** de sistemas operativos de IEEE
- **Objetivo**: ofrecer portabilidad de las aplicaciones entre diferentes plataformas y sistemas operativos
- **No** es una implementación, solo define una interfaz (visión)
- Diferentes estándares:
  - 1003.1 → Servicios básicos del SO
  - 1003.1a → Extensiones a los servicios básicos
  - 1003.1b → Extensiones de tiempo real
  - 1003.1c → Extensiones de procesos ligeros
  - 1003.2 → Shell y utilidades
  - 1003.2b → Utilidades adicionales

UNIX03

El Single Unix Specification UNIX 03 es una evolución que engloba a POSIX y otros estándares (x/Open XPG4, ISO C). Incluye no solamente la interfaz de programación, sino también otros aspectos:

- Servicios ofrecidos
- Intérprete de mandatos
- Utilidades disponibles

Características de POSIX

- **Nombres** de funciones **cortos** y en letras minúsculas: fork, read, close, …
- Las funciones normalmente **devuelven 0** en caso de **éxito** o **-1** en caso de **error** (en general, aunque pueden darse otros cosos)
  - Puede indicarse el error también en la variable e "errno"
- Los recursos gestionados por el sistema operativo se referencian mediante **descriptores** (número enteros), lo veremos más adelante.

2 Servicios de procesos

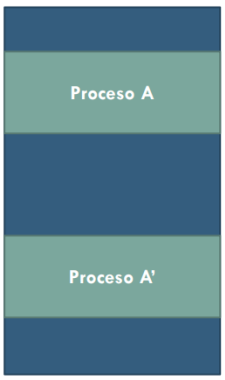
Ejemplo de ejecución de servicios

```
#include <sys/types.h>
#include <stdio.h>
  int main(int argc, char** argv) {
    pid_t pid;
    pid = fork();
    switch (pid) {
      case -1: /* error */
        exit(-1);
      case 0: /* proceso hijo */
        if (execvp(argv[1], &argv[1])<0) { perror("error"); }
          break;
      default:
        printf("Proceso padre");
    }
    return 0;
  }
```

Servicio fork → (PUEDE CAER EN EL EXAMEN)

Es una instrucción que **duplica el proceso** que **invoca la llamada**. Esto se realiza con el objetivo de crear procesos manteniendo una estructura de padres e hijos, pudiendo saber de dónde viene cada proceso a partir del proceso principal (kernel). Tiene la sintaxis **pid_t fork (void);**.

- **Proceso de ejecución:**
  - Se ejecuta la función fork, que hace una llamada syscall, mediante una instrucción trap, dejando suspendido el proceso que llama al fork
  - Se hace la copia del proceso con su estado exactamente igual que el primero. Para identificarlos, se analizará el valor que devuelve la función fork, que podrá ser:
    - Si es el **proceso padre** (el original): devuelve un valor distinto de 0, que es el identificador del hijo y se almacena fuera del proceso
    - Si es el **proceso hijo** (la copia): devuelve el valor 0
    - En caso de error se devolverá -1



  - Se retorna al código del proceso principal (después de la ejecución del fork)
- El proceso padre y el proceso hijo siguen ejecutando el mismo programa

- El proceso hijo hereda los ficheros abiertos del proceso padre, y se copian los descriptores de archivos abiertos
- Se desactivan las alarmas pendientes

Servicio exec

Este servicio se encarga de reemplazar todo el código del proceso que llama al servicio por el código de otro archivo ejecutable, introducido como parámetro en la llamada al servicio.

- Es un servició único pero con múltiples funciones de biblioteca. Sintaxis:
  - `int execl(const char *path, const char *arg, ...);`
  - `int execv(const char* path, char* const argv[]);`
  - `int execve(const char* path, char* const argv[], char* const envp[]);`
  - `int execvp(const char *file, char *const argv[])`
- Cambia la imagen del proceso actual a:
  - `path` → ruta al archivo ejecutable
  - `file` → busca el archivo ejecutable en todos los directorios especificados por PATH
- Descripción
  - Devuelve -1 en caos de error, en caso contrario no retorna (ya que se ha reemplazado el código)
  - El mismo proceso es el que ejecuta el otro programa (ya que como hemos dicho se ha reemplazado por el ejecutable llamado por el servicio)
  - Los ficheros abiertos permanecen abiertos
  - Las señales con la acción por defecto seguirán por defecto, las señales con manejador tomarán la acción por defecto

Servicio exit

Este servicio finaliza la ejecución de un proceso. Su sintaxis es **void exit (status)**. Funciones:

- Se cierran todos los descriptores de ficheros abiertos
- Se liberan todos los recursos del proceso
- Se libera el BCP (bloque de control de procesos) del proceso

3 Operaciones con ficheros

Operaciones genéricas sobre ficheros

- **Crear**: crea un fichero con un nombre y unos atributos
- **Borrar**: borra un fichero a partir de su nombre
- **Abrir**: abre un fichero a partir de su nombre para permitir operaciones de acceso
- **Cerrar**: cierra un fichero abierto
- **Leer**: lee un dato de un fichero abierto a aun almacén en memoria
- **Escribir**: escribe datos a un fichero abierto desde un almacén en memoria
- **Posicionar**: mueve el apuntador usado para acceder al fichero, afectando a operadores posteriores
- **Control**: permite manipular los archivos de un fichero

Servicios POSIX para ficheros

- **Visión lógica** → conjunto de servicios para manipular ficheros
- Se mantiene un puntero asociado a cada fichero abierto: el puntero indica la posición a partir de la cual se realizará la siguiente operación → descriptor
- La mayor parte de las operaciones trabajan con **descriptores de ficheros**
  - Es un número entero entre 0 y 64K, que identifica a cada fichero abierto por un proceso
  - Se obtiene al abrir el fichero (mediante la operación **open**)
  - El resto de las operaciones identifican a un fichero por su descriptor
- **Descriptores predefinidos** → aquellos que ya apuntan a algún sitio sin realizar operaciones:
  - 0 → entrada estándar: generalmente correspondiente al teclado
  - 1 → salida estándar: generalmente la pantalla
  - 2 → salida de error: generalmente la pantalla al igual que el 1

Cada proceso tiene una tabla de ficheros abiertos; es decir, una tabla de descriptores de ficheros. Cuando se duplica un proceso mediante el servicio **fork**:

- Se duplica la tabla de archivos/ficheros abiertos
- Se comparte la tabla intermedia de nodos-i y posiciones; la que guarda los datos del archivo abierto

Protección

La protección de los ficheros hace referencia al **conjunto de permisos** que un usuario o grupo de usuarios tiene sobre el mismo. Esta protección viene codificada por un número en octal de 9 dígitos siguiendo la siguiente estructura:

- Especial Propietario grupo mundo
  - rwx rwx rwx
- Significado: r → read / w → write / x → ejecución
- Aquellos campos en los que el usuario no tenga permisos se representan mediante un guión. Ejemplo: -rwxr-xr-x → Solo el propietario tiene permiso de escritura y los demás tienen permisos de lectura y ejecución
- Estos permisos se pueden representar, como hemos dicho antes, mediante un número en octal. De este modo, el ejemplo anterior se puede escribir como 0755 (el 0 inicial indica que el formato es octal)

Ficheros y directorios POSIX

Existen diferentes **tipos de ficheros**:

- Normales
- Directorios
- Especiales

Los **nombres de fichero y directorios** siguen la siguiente sintaxis:

- Nombre completo (empieza por /) → Ejemplo: **`/usr/include/stdio.h`**
- Nombre relativo al directorio actual (sin /) → Ejemplo: **`stdio.h`** asumiendo que /usr/include es el directorio actual
- La entrada . hace referencia al directorio donde nos encontramos, y la entrada .. indica al directorio padre del que estamos→ Ejemplo: ../include/stdio.h

SERVICIOS DE FICHEROS

CREAT → Creación de fichero

- **Descripción**:
  - El fichero se abre para escritura
  - Si no existe el fichero, crea un fichero vacio
  - Si existe el fichero, lo trunca (borra todo el contenido) sin cambiar los bits de permiso
- **Sintaxis** (los include's no hace falta aprenderlos para el examen)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (char *name, mode_t mode);
```

- **Argumentos**:
  - `name` → nombre del fichero que queremos crear
  - `mode` → Bits de permiso a establecer en el fichero. Estos bits de codifican en octal tal y como se explica en el apartado anterior "protección"
- **Devuelve**:
  - La función devuelve un **descriptor de fichero** si se crea correctamente ó **-1** en caso de error
- Ejemplo → `fd = creat ("datos.txt", 0751);` → Almacena en 'fd' el descriptor del fichero creado "datos.txt", y crea este fichero con los permisos rwxr-x--x

UNLINK – Borrado de fichero

- **Descripción**:
  - Decrementa el contador de enlaces del fichero. Si el contador llega a 0, borra el fichero y libera sus recursos.
  - Aclaración: el mismo fichero puede tener asociado varios nombres, a partir de los cuales podemos llamar o acceder al fichero. Esta relación entre el mismo fichero y sus nombres se denomina "enlace". El conjunto de enlaces se guarda en una tabla o contador de enlaces, que almacena la relación nombre-fichero. Al hacer unlink, borramos dicho enlace entre un nombre y su fichero; y cuando al borrar dicho enlace

no queda ninguna relación de un nombre con el fichero, este se elimina, liberando el espacio y recursos que ocupaba.

- **Sintaxis:** (los include's no hace falta aprenderlos para el examen)

```
#include <unistd.h>
int unlink (const char* path); //
```

Se declara el argumento como constante, para indicar al usuario que la path pasada no se va a modificar

- **Argumentos:**
  - `path` → nombre del fichero que se va a desenlazar. Debe seguir la sintaxis establecida anteriormente en el apartado "Ficheros y directorios POSIX"
- **Devuelve**:
  - 0 si se ha ejecutado correctamente ó -1 si ha habido un error

OPEN – Apertura de fichero

- **Descripción**:
  - Servicio utilizado para abrir un fichero, es decir, para poder leer, escribir o ejecutarlo
  - Abrir el fichero no implica visualizar su contenido, sino prepararlo para operar con el
  - Este servicio cuenta con distintos flags utilizados para establecer las condiciones de apertura y decisiones en dicha acción, especificadas a continuación
- **Sintaxis**: (los include's no hace falta aprenderlos para el examen)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (char *name, int flag, …);
```

- **Argumentos**:
  - `name` → puntero al nombre del fichero a abrir. Debe seguir la sintaxis establecida anteriormente en el apartado "Ficheros y directorios POSIX"
  - `flags` → opciones/condiciones de apertura. Se pueden indicar varios flags en el mismo open. Tipos:
    - O_RDONLY → se abre el fichero en solo lectura
    - O_WRONLY → se abre el fichero en solo escritura
    - O_RDWR → se abre el fichero para lectura y escritura
    - O_APPEND → el puntero de acceso se desplaza al final del fichero abierto, de modo que al escribir sobre el fichero, se mantendrá el contenido y se escribirá al final
    - O_CREAT → Si existe el fichero no hace nada, en cambio, si no existe lo crea, con los permisos indicados en la instrucción
    - O_TRUNC → Trunca si se abre para escritura; es decir, borra el contenido y escribe desde el inicio
- **Ejemplo**:
  - `fd = open ("/home/juan/datos.txt", O_WRONLY | O_CREAT | O_TRUNC, 0750);` → Abre el fichero de la ruta indicada, en solo lectura, estableciendo que si no

existe se cree, que se trunque el contenido del fichero, y estableciendo en caso de que se tenga que crear, los permisos codificados en octal 0750 (rwxr-----)

CLOSE – Cierre de fichero

- **Descripción**:
  - Se cierra un fichero; es decir, se elimina la asociación entre el proceso y el fichero que estaba abierto, liberando el descriptor asociado al fichero para utilizarlo en otro momento, con el mismo u otro fichero
- **Sintaxis**:

```
int close (int fd);
```

- **Argumentos**:
  - fd → descriptor del fichero a cerrar
- **Devuelve**:
  - 0 si se ha realizado correctamente ó -1 si se ha producido un error

READ – Lectura de fichero

- **Descripción**:
  - Servicio para leer el contenido de un fichero
  - En una lectura se transfieren (leen) n_bytes. Pueden leerse menos datos de los solicitados si se llega antes al final del fichero o se interrumpe la lectura por una señal
  - Después de una lectura, se incrementa el puntero del fichero con el número de bytes realmente transferidos (leídos)
- **Sintaxis**:

```
#include <sys/types.h>
ssize_t read (int fd, void *buf, size_t n_bytes);
```

- **Argumentos**:
  - `fd` → descriptor del fichero a leer
  - `buf` → zona donde se van a almacenar los datos; es decir, una cadena de caracteres
  - `n_bytes` → número de bytes a leer
- **Devuelve**:
  - En `ssize_t` (que puede ser cualquier variable int) devuelve el número de bytes realmente leídos; ó -1 si se ha producido un error

WRITE – Escritura de fichero

- **Descripción**:
  - Servicio para escribir en un fichero
  - Transfiere n_bytes. Puede escribir menos datos de los solicitados si se alcanza el tamaño máximo de un fichero o se interrumpe por una señal
  - Después de la escritura se incrementa el puntero del fichero con el número de bytes realmente transferidos
  - Si se rebasa el fin del fichero, el fichero aumenta de tamaño

- **Sintaxis**:

```
#include <sys/types.h>
ssize write (int fd, void *buf, size_t n_bytes);
```

- **Argumentos**:
  - fd → descriptor del fichero donde se va a escribir
  - buf → zonda de datos que se van a escribir en el fichero
  - n_bytes → número de bytes a escribir
- **Devuelve**:
  - En `ssize_t` (que puede ser cualquier variable int) devuelve el número de bytes realmente escritos; ó -1 si se ha producido un error

LSEEK – Movimiento del puntero de posición

- **Descripción**:
  - Servicio utilizado para recolocar el puntero de posición de un fichero, es decir, el puntero que indica la posición del fichero donde se va a leer o escribir.
  - Coloca el puntero de acceso asociado a `fd`
  - La nueva posición se calcula según lo puesto en la base de desplazamiento `whence`
- **Sintaxis**:

```
#include <sys/types.h>
#include <unistd.h>
off_l lseek (int fd, off_t offset, int whence);
```

- **Argumentos**:
  - `fd` → descriptor del fichero
  - `offset` → desplazamiento del puntero
  - `whence` → base del desplazamiento. Esta puede tomar los valores:
    - SEEK_SET → posición del puntero = offset
    - SEEK_CUR → posición del puntero = posición actual + offset
    - SEEK_END → posición del puntero = tamaño del fichero (posición final) + offset
- **Devuelve**:
  - En `off_t` (que puede ser cualquier variable int) devuelve la nueva posición del puntero ó -1 si se ha producido un error

FNCTL – Modificación de atributos (no es importante)

- **Descripción**:
  - Modifica los atributos de un fichero abierto
- **Sintaxis**:

```
#include <sys/types.h>
int fnctl (int fildes, int cmd /* arg*/ …);
```

- **Argumentos:**
  - `fildes` → descriptor del fichero
  - `cmd` → mandato para modificar atributos, puede haber varios

- **Devuelve**:
  - Si se realiza correctamente 0, ó -1 si se produce un error

DUP – Duplicación de descriptor de fichero

- **Descripción**:
  - Crea un nuevo descriptor de fichero que tiene en común con el anterior:
    - Acceden ambos al mismo fichero
    - Comparten el mismo puntero de posición
    - El modo de acceso es idéntico
  - El nuevo descriptor tendrá el menor valor numérico posible
- **Sintaxis**:

```
int dup (int fd)
```

- **Argumentos**:
  - `fd` → descriptor del fichero a duplicar
- **Devuelve**:
  - Devuelve el descriptor de fichero nuevo que comparte todas las propiedades del fd; o -1 en caso de error.
- **Variación**:
  - Mediante `int dup2 (int fd, int fn)`, se crea un nuevo descriptor, con valor `fn` que comparte las propiedades del descriptor `fd`. En el caso de que el descriptor `fn` ya exista, lo borra y después crea el nuevo descriptor.

FTRUNCATE – Asignación de espacio a un fichero

- **Descripción**:
  - Establece como nuevo tamaño del fichero el valor `length`. Si `length` es 0, se trunca (limpia) el fichero
- **Sintaxis**:

```
#include <unistd.h>
int ftruncate (int fd, off_t length);
```

- **Argumentos**:
  - `fd` → descriptor del fichero
  - `length` → nuevo tamaño del fichero
- **Devuelve**:
  - Devuelve 0 si se ha realizado correctamente, ó -1 en caso de error

STAT – Información sobre un fichero

- **Descripción**:
  - Obtiene información sobre un fichero y la almacena en una estructura de tipo struct stat. Esta estructura no hace falta definirla, sino que se llama directamente invocando a la instrucción `struct stat nombre`. Contenido de la estructura:

```
struct stat {
    mode_t st_mode; /* modo del fichero */
    ino_t st_ino; /* número del fichero */
    dev_t st_dev; /* dispositivo */
    nlink_t st_nlink; /* número de enlaces */
    uid_t st_uid; /* UID del propietario */
    gid_t st_gid; /* GID del propietario */
    off_t st_size; /* número de bytes */
    time_t st_atime; /* último acceso */
    time_t st_mtime; /* última modificacion */
    time_t st_ctime; /* último modificacion de datos */ };
```

- **Sintaxis**:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(char *name, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

- **Argumentos**:
  - `name` → nombre del fichero a analizar (si se usa stat, se analiza el fichero identificándolo por su nombre)
  - `fd` → descriptor del fichero a analizar (si se usa fstat, se analiza el fichero identificándolo por su descriptor)
  - `buf` → Puntero a un objeto de tipo `struct stat` antes declarado donde se almacenará la información sobre el fichero
- **Devuelve**:
  - Devuelve 0 si se ha realizado correctamente la operación, ó –1 en caso de error
- **Comprobación del tipo de fichero** analizando el st_mode:
  - S_ISDIR (buf.st_mode) → Verdadero si el fichero es un directorio
  - S_ISCHR (buf.st_mode) → Verdadero si es un fichero especial de caracteres
  - S_ISBLK (buf.st_mode) → Verdadero si especial de bloques
  - S_ISREG (buf.st_mode) → Verdadero si es un fichero normal
  - S_ISFIFO (buf.st_mode) → Verdadero si es un pipe (tubería) o un FIFO

UTIME – Alteración de atributos de fecha

- **Descripción**:
  - Cambia las fechas de último acceso y última modificación según los valores de la estructura `struct utimbuf`
- **Sintaxis**:

```
#include <sys/stat.h>
#include <utime.h>
int utime(char *name, struct utimbuf *times);
```

- **Argumentos**:
  - `name` → nombre del fichero

- `times` → estructura utimbuf con las fechas de ultimo acceso y modificación:
  - time_t actime → fecha de acceso
  - time_t mctime → fecha de última modificación

➢ **Devuelve**:
- 0 en caso de que se haya realizado correctamente ó -1 en caso de error

Ejemplo copia de un fichero a otro

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

#define BUFSIZE 512

main(int argc, char **argv) {
  int fd_ent, fd_sal;
  char buffer[BUFSIZE];
  int n_read;

  /* Se abre el fichero de entrada */
  fd_ent = open(argv[1],
    O_RDONLY);
  if (fd_ent < 0) {
    perror("open");
    exit(-1);
  }

  /* Se crea el fichero de salida */
  fd_sal = creat(argv[2], 0644);
  if (fd_sal < 0) {
    close(fd_ent);
    perror("open");
    exit(-1);
  }
  /* Se ejecuta el bucle de lectura
  del fichero de entrada */
  while ((n_read = read(fd_ent,
    buffer, BUFSIZE)) > 0) {
    /* Se escribe el buffer al
    fichero de salida */
    if (write(fd_sal, buffer,
    n_read) < n_read) {
      perror("write2);
      close(fd_ent); close(fd_sal);
      exit(-1);
    }
  }

  /* Se controla si al leer se
  devuelve un numero negativo, es
  decir, da error */
  if (n_read < 0) {
    perror("read");
    close(fd_ent); close(fd_sal);
    exit(-1);
  }

  close(fd_ent); close(fd_sal);
  exit(0);
}
```

Servicios POXIX para directorios

➢ **Visión lógica:**
- Un directorio es un fichero con registros tipo "estructura DIR"
- Por tanto se puede operar como un fichero, pero no se puede escribir desde un programa, solo se puede leer

➢ **Estructura DIR**
- `d_ino`; → Nodo_i
- `d_off`; → Posición en el fichero del elemento del directorio
- `d_reclen`; → Tamaño del directorio
- `d_type;` → Tipo de elemento
- `d_name[0];` → Nombre del fichero **de longitud variable**

- Al ser el nombre de longitud variable no se puede manipular como registros de longitud fija.
- Se trabajará mediante llamadas al sistema para manejar directorios

Servicios

- `DIR *opendir(const char *dirname);`
  - Abre el directorio y devuelve un puntero al principio de tipo DIR
- `int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);`
  - Lee la siguiente entrada de directorio y la devuelve en una struct dirent
- `long int telldir(DIR *dirp);`
  - Indica la posición actual del puntero dentro del archivo del directorio
- `void seekdir(DIR *dirp, long int loc);`
  - Avanza desde la posición actual hasta la indicada en "loc". Nunca saltos atrás.
- `void rewinddir(DIR *dirp);`
  - Resetea el puntero del archivo y lo pone otra vez al principio
- `int closedir(DIR *dirp);`
  - Cierra el archivo del directorio

Proyección en POSIX

- **Descripción**:
  - Establece una proyección entre el espacio de direcciones de un proceso y un archivo
  - Es decir, consiste en proyectar una porción de un fichero (de disco) en una nueva región de memoria principal, para poder acceder mediante memoria virtual, de forma que se puede utilizar espacio físico del disco como si fueran páginas de memoria RAM
- **Sintaxis**:
  - `void *mmap(void *direc, size_t length, int prot,int flags, int fd, off_t desp);`
- **Argumentos**:
  - `direc` → dirección donde proyectar. Si es NULL, el SO elige la dirección
  - `length` → especifica el número de bytes a proyectar
  - `prot` → protección para la zona (se pueden concatenar con |). Tipos:
    - PROT_READ: Se puede leer
    - PROT_WRITE: Se puede escribir
    - PROT_EXEC: Se puede ejecutar
    - PROT_NONE: No se puede acceder a los datos
  - `flags` → propiedades de la región. Tipos:
    - MAP_SHARED: La región es compartida. Las modificaciones afectan al fichero y los procesos hijos comparten la misma región
    - MAP_PRIVATE: La región es privada. El fichero no se modifica y los procesos hijos obtienen duplicados no compartidos
    - MAP_FIXED: El fichero debe proyectarse en la dirección especificada por la llamada)
  - `fd` → descriptor del fichero que se desea proyectar en memoria

- `desp` → desplazamiento inicial sobre el archivo

➢ **Devuelve**

- La dirección de memoria donde se ha proyectado el archivo

Desproyección en POSIX

➢ **Descripción**:

- Des proyecta la parte del espacio de direcciones de un proceso desde la dirección `direct` hasta `direct + lon`

➢ **Sintaxis**:

- `void munmap(void *direc, size_t lon);`

Ejemplo → Contar número de blancos de un fichero con mmap

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

int main() {
  int fd;
  struct stat dstat;
  int i, n;
  char c,
  char * vec;

  fd = open("datos.txt",O_RDONLY);
  fstat(fd, &dstat);
  vec = mmap(NULL, dstat.st_size,
    PROT_READ, MAP_SHARED, fd, 0);
  close(fd);

  c =vec;
  for (i=0;i<dstat.st_size;i++) {
    if (*c==' ') {
      n++;
    }
    c++;
  }

  munmap(vec, dstat.st_size);
  printf("n=%d,\n",n);
  return 0;
}
```

Ejemplo → Copia de un fichero

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

int main() {
  int i, fd1, fd2;
  struct stat dstat;
  char * vec1, *vec2, *p, *q;

  fd1 = open("f1", O_RDONLY);
  fd2 = open("f2",
    O_CREAT|O_TRUNC|O_RDWR,0640);
  fstat(fd1,&dstat);
  ftruncate(fd2, dstat.st_size);
  vec1=mmap(0, bstat.st_size,
    PROT_READ, MAP_SHARED, fd1,0);
  vec2=mmap(0, bstat.st_size,
    PROT_READ, MAP_SHARED, fd2,0);

  close(fd1); close(fd2);
  p=vec1; q=vec2;

  for (i=0;i<dstat.st_size;i++) {
    *q++ = *p++;
  }

  munmap(fd1, bstat.st_size);
  munmap(fd2, bstat.st_size);
  return 0;
}
```