

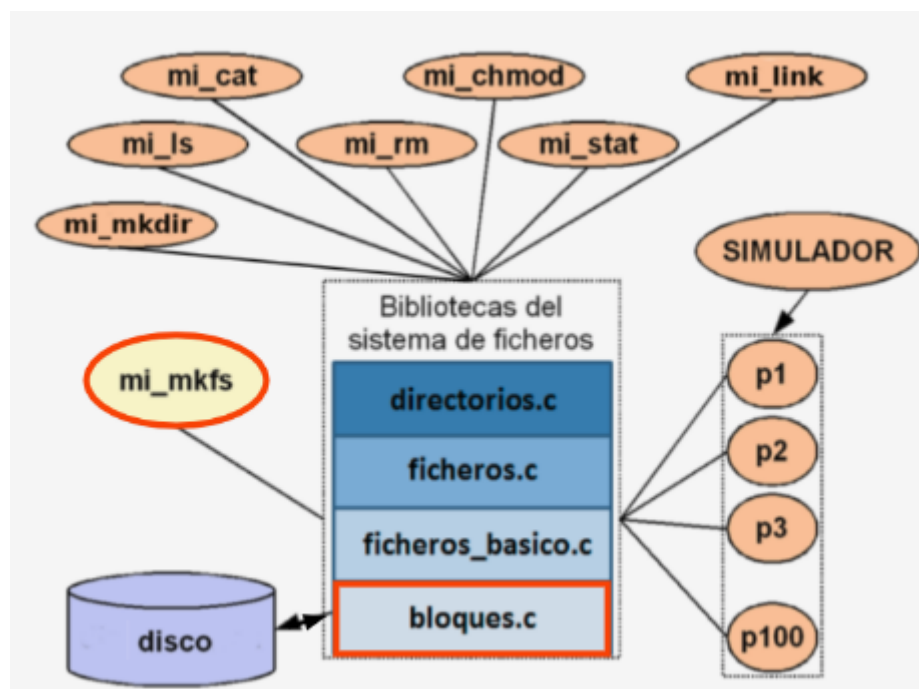
## Nivel 1: bloques.c {bmount(), bumount(), bwrite(), bread()} y mi\_mkfs.c

Un **fichero lógico** se compone de una secuencia de **bloques lógicos**. Los bloques lógicos son **consecutivos** (aunque pueden haber “huecos”).

El **tamaño del bloque lógico** se corresponde con el **tamaño del bloque físico**, y es el mismo para todo el sistema de ficheros (**BLOCKSIZE**) y suele ser múltiplo de 512 bytes<sup>1</sup>. En nuestro dispositivo virtual los bloques serán de 1024 bytes.

A cada bloque lógico ocupado de un fichero le corresponderá un **bloque físico** del dispositivo virtual, que puede estar **en cualquier lugar de la zona de datos**. Bloques lógicos contiguos no tienen porqué ocupar bloques físicos contiguos.

Vamos a programar en **bloques.c** las funciones básicas para **montar** y **desmontar** el dispositivo virtual y para **leer/escribir un bloque** en él. Luego implementaremos la primera versión de **mi\_mkfs.c** que será el programa que **formateará** nuestro dispositivo virtual, instalando el sistema de ficheros de **nbloques** (parámetro pasado por consola), llamando a las funciones de **bloques.c** para montar/desmontar el dispositivo virtual e inicializar a 0s sus **nbloques**.



<sup>1</sup> El tamaño elegido para el bloque va a influir en las prestaciones globales del sistema.

- Bloques grandes  $\Rightarrow$  velocidad de transferencia grande entre el disco y la memoria.
- Si demasiado grandes  $\Rightarrow$  capacidad de almacenamiento del disco desaprovechada cuando abundan los archivos pequeños que no llegan a ocupar un bloque completo (fragmentación interna)..

## bloques.c

### 1) int bmount(const char \*camino);

Función para montar el dispositivo virtual, y dado que se trata de un fichero, esa acción consistirá en abrirlo.

Lamará a la función `open()` del sistema, pasándole el camino que ha recibido como parámetro, para obtener el **descriptor del fichero** que usaremos como dispositivo virtual.

```
int open(const char *camino, int oflags, mode_t mode);
```

El nombre del fichero va en el argumento de la función (nos lo proporcionará el administrador del sistema desde consola cuando inicialice el sistema de ficheros a través de `mi_mkfs`).

Valores para oflags:

<code>O_RDONLY</code>	Abre el fichero sólo para lectura
<code>O_WRONLY</code>	Abre el fichero sólo para escritura
<code>O_RDWR</code>	Abre el fichero para lectura y escritura
<code>O_APPEND</code>	Añade información al final del fichero
<code>O_TRUNC</code>	Inicialmente borra todos los datos del fichero
<code>O_CREAT</code>	Si el fichero no existe lo crea. En este caso se requiere el 3er parámetro (permisos).
<code>O_EXCL</code>	Combinado con la opción <code>O_CREAT</code> , asegura que el que lo llama debe crear el fichero, si ya existe la llamada fallará.

Podemos abrirlo como `O_RDWR | O_CREAT`.

Los permisos se representan en octal, por ej: `0666` significa que damos permiso de lectura (r) y escritura (w) a usuario, grupo y otros: rw-rw-rw-. [[+Información sobre permisos](#)].

octal	binario	permiso
7	111	rwX
6	110	rw-
5	101	r-X
4	100	r--
3	011	-wX
2	010	-w-
1	001	--X
0	000	---

Un proceso o programa tiene asignada una **tabla de descriptores de ficheros** (nº entero de 0-19, 0: entrada estándar, 1: salida estándar, 2: salida de error estándar). La función `open()` retornará el **descriptor del fichero** (el más bajo libre en la tabla de descriptores) para ser usado en las siguientes operaciones de E/S.

Para abrir un fichero se realiza una **llamada al sistema** y puede producir errores que hay que gestionar [[control de errores en las llamadas al sistema](#)]:

```
if (descriptor == -1) {  
    ... //error  
}
```

`bmount()` devuelve -1 si ha habido error<sup>2</sup>, o el **descriptor** obtenido si ha ido bien.

### 2) int bumount();

Desmonta el dispositivo virtual. Básicamente llama a la función `close()` para liberar el descriptor de fichero.

```
int close(int descriptor);
```

La función `bumount()` devuelve 0 si se ha cerrado el fichero correctamente, o -1 en caso contrario. Podéis utilizar los mismos símbolos que en el caso anterior.

### 3) int bwrite(unsigned int nbloque, const void \*buf);

Escribe 1 bloque en el dispositivo virtual, en el **bloque físico** especificado por `nbloque`.

Para ello volcará el contenido de un *buffer* de memoria (que tendrá el tamaño de un bloque), apuntado por `*buf`, en la posición (nº de byte) del dispositivo virtual correspondiente al nº de bloque, `nbloque`, especificado en el argumento.

Primeramente, con la función `lseek()`, movemos el puntero del fichero en el *offset* correcto, para ello calculamos el desplazamiento dentro del dispositivo virtual donde hay que escribir:

nº de bloque \* tamaño de bloque (o sea `nbloque * BLOCKSIZE`),

---

<sup>2</sup> En lugar de utilizar el símbolo predefinido `EXIT_FAILURE` o -1 para situaciones de ERROR, podéis crear un símbolo propio en `bloques.h`, por ejemplo `FALLO`. Lo importante es que si se trata de un error devuelva como valor -1 y no 1 (como tiene definido `EXIT_FAILURE`), ya que habrá situaciones en que el valor 1 sea un valor de retorno que no implique error. Para utilizar en casos de retorno de éxito podéis por ejemplo crear el símbolo `EXITO` definido con valor 0.

comenzando a contar, como punto de referencia, desde el inicio del fichero: `SEEK_SET`. Cuando ya esté el puntero posicionado, entonces se utilizará la función `write()` para volcar el contenido del buffer (de tamaño `BLOCKSIZE`) en dicha posición del dispositivo virtual.

```
off_t lseek(int descriptor, off_t desplazamiento, int punto_de_referencia);
```

```
size_t write(int descriptor, const void *buf, size_t nbytes);
```

La función `bwrite()` devuelve el nº de bytes que ha podido escribir (si ha ido bien, será `BLOCKSIZE`), o -1 (o `FALLO`) si se produce un error.

#### 4) `int bread(unsigned int nbloque, void *buf);`

Lee 1 bloque del dispositivo virtual, que se corresponde con el **bloque físico** especificado por `nbloque`.

Para ello se volcará en un *buffer* de memoria (que tendrá el tamaño de un bloque), apuntado por `*buf`, los `nbytes` (`BLOCKSIZE`) contenidos a partir de la posición (nº de byte) del dispositivo virtual correspondiente al nº de bloque, `nbloque`, especificado en el argumento.

Primeramente, con la función `lseek()`, movemos el puntero del fichero en el *offset* correcto, para ello calculamos el desplazamiento dentro del dispositivo virtual donde hay que escribir:

nº de bloque \* tamaño de bloque (o sea `nbloque * BLOCKSIZE`),

comenzando a contar, como punto de referencia, desde el inicio del fichero: `SEEK_SET`. Cuando ya esté el puntero posicionado, entonces se utilizará la función `read()` para volcar en el buffer (de tamaño `BLOCKSIZE`) el contenido de los `nbytes` (`BLOCKSIZE`) a partir de dicha posición del dispositivo virtual.

```
off_t lseek(int descriptor, off_t desplazamiento, int punto_de_referencia);
```

```
size_t read(int descriptor, void *buf, size_t nbytes);
```

La función `bread()` devuelve el nº de bytes que ha podido leer (si ha ido bien, será `BLOCKSIZE`), o -1 (o `FALLO`) si se produce un error.

#### Observaciones:

- Es conveniente que el descriptor del fichero se almacene como variable global estática (para que sólo pueda ser accedida en `bloques.c`).

```
static int descriptor = 0;
```

- El tamaño de bloque, `BLOCKSIZE`, ha de ser una constante, en nuestro caso 1024.
- En `bloques.h` hay que incluir las cabeceras básicas fundamentales, y la declaración de funciones y constantes:

```
// bloques.h

#include <stdio.h> //printf(), fprintf(), stderr, stdout, stdin
#include <fcntl.h> //O_WRONLY, O_CREAT, O_TRUNC
#include <sys/stat.h> //S_IRUSR, S_IWUSR
#include <stdlib.h> //exit(), EXIT_SUCCESS, EXIT_FAILURE, atoi()
#include <unistd.h> //SEEK_SET, read(), write(), open(), close(), lseek()
#include <errno.h> //errno
#include <string.h> //strerror()

#define BLOCKSIZE 1024 // bytes

#define EXITO 0 //para gestión errores
#define FALLO -1 //para gestión errores

int bmount(const char *camino);
int bumount();
int bwrite(unsigned int nbloque, const void *buf);
int bread(unsigned int nbloque, void *buf);
```

- En `bloques.c` (y en `mi_mkfs.c`) tenéis que incluir luego esta cabecera:

```
#include "bloques.h"
```

---

## mi\_mkfs.c

Vamos a escribir una primera versión del programa `mi_mkfs.c` que utilice las funciones de `bloques.c`:

- El programa `mi_mkfs.c` sirve para formatear el dispositivo virtual con el tamaño adecuado de bloques, `nbloques`. Debe ser llamado desde la línea de comandos con los siguientes parámetros para dar nombre al dispositivo virtual y determinar la cantidad de bloques de que dispondrá nuestro sistema de ficheros:

**\$ ./mi\_mkfs <nombre\_dispositivo> <nbloques>**

Recordemos que los parámetros de consola se recuperan en el *main* con:

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

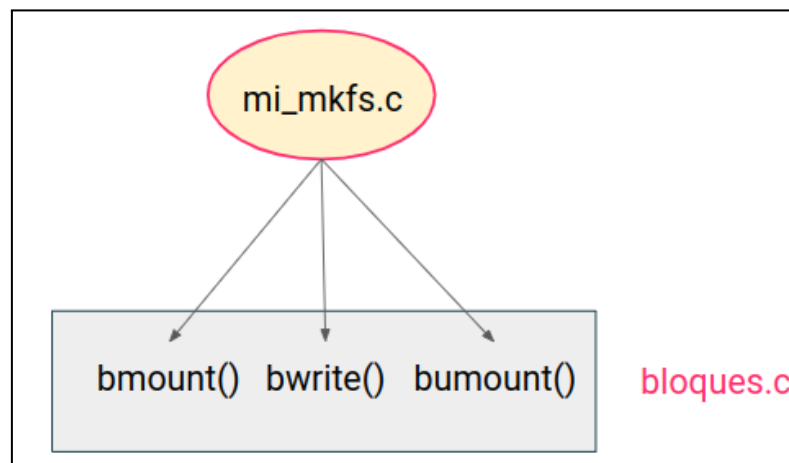
donde:

**argc**: número de parámetros

argc=3

**argv**: vector de punteros a los parámetros:

- argv[0]="mi\_mkfs"
  - argv[1]=nombre\_dispositivo
  - argv[2]=nbloques (puede sernos útil la función *atoi()* para obtener el valor numérico a partir del string)
- El primer paso es montar (*bmount()*) el dispositivo virtual (en nuestro caso será abrir el fichero que utilizamos para tal fin).
  - Es necesario llamar a *bwrite()* el número de veces necesario (indicado por el valor de *nbloques*, pasado como parámetro) para inicializar a 0s el fichero usado como dispositivo virtual.
  - En este caso, el *buffer* de memoria empleado puede ser un array de tipo *unsigned char* del tamaño de un bloque (lo inicializaremos a 0s con la función *memset()*, y hay que incluir *<string.h>* para utilizarla).
  - El último paso es desmontar (*bumount()*) el dispositivo virtual, en nuestro caso cerrar el fichero que utilizamos para tal fin.



Esquema de llamadas a funciones desde mi\_mkfs.c

### Observaciones:

- A cada función que implementéis hay que asociarle un comentario que explique para qué sirve, qué son cada uno de los parámetros de entrada, y qué devuelve. Opcionalmente también se añadirá un listado de a qué funciones llama y por cuáles es llamada (eso se irá completando a lo largo del desarrollo de la práctica)
- De momento podéis compilar conjuntamente **mi\_mkfs.c** y **bloques.c** desde la línea de comandos con **gcc** de la siguiente manera:

```
$ gcc -o mi_mkfs mi_mkfs.c bloques.c
```

- Y luego ejecutarlo pasándole los parámetros correspondientes

```
$ ./mi_mkfs <nombre_dispositivo> <nbloques>
```

- Pero es más recomendable utilizar desde el principio un **Make**<sup>3</sup> (será imprescindible a medida que vayamos creando más y más programas)

```
CC=gcc
CFLAGS=-c -g -Wall -std=gnu99
#LDFLAGS=-pthread

SOURCES=mi_mkfs.c bloques.c #ficheros_basico.c leer_sf.c ficheros.c escribir.c leer.c
truncar.c permitir.c directorios.c mi_mkdir.c mi_chmod.c mi_ls.c mi_link.c mi_escribir.c
mi_cat.c mi_stat.c mi_rm.c semaforo_mutex_posix.c simulacion.c verificacion.c
LIBRARIES=bloques.o #ficheros_basico.o ficheros.o directorios.o
semaforo_mutex_posix.o
INCLUDES=bloques.h #ficheros_basico.h ficheros.h directorios.h
semaforo_mutex_posix.h simulacion.h
PROGRAMS=mi_mkfs #leer_sf escribir leer truncar permitir mi_mkdir mi_chmod mi_ls
mi_link mi_escribir mi_cat mi_stat mi_rm simulacion verificacion
OBJS=$(SOURCES:.c=.o)

all: $(OBJS) $(PROGRAMS)

$(PROGRAMS): $(LIBRARIES) $(INCLUDES)
    $(CC) $(LDFLAGS) $(LIBRARIES) $@.o -o $@

%.o: %.c $(INCLUDES)
    $(CC) $(CFLAGS) -o $@ -c $<
```

<sup>3</sup> Si hacéis un *copy&paste* del código del makefile, recordad que hay que sustituir los espacios en blanco del sangrado por un tabulador

```
.PHONY: clean  
clean:  
rm -rf *.o *~ $(PROGRAMS) disco* ext*
```

- Para depurar el código, es recomendable utilizar [gdb](#) (o alguna de sus variantes gráficas: [Nemiver](#), por ejemplo) o utilizar las posibilidades de depuración que ofrece el editor [Visual Studio Code](#).
- Para analizar el contenido del fichero usado como dispositivo virtual, es recomendable utilizar cualquier editor hexadecimal: [Okteta](#), o [GHex](#), por ejemplo.

## Tests de prueba

```
$ ./mi_mkfs disco 100000
```

```
$ ls -l disco #ha de ocupar 102.400.000 bytes y tener permisos rw-rw-rw-4
```

```
-rw-rw-rw- 1 uib uib 102400000 de febr. 22 11:13 disco
```

También podéis comprobar el tamaño mirando las propiedades del fichero en el sistema de ventanas <sup>5</sup>:

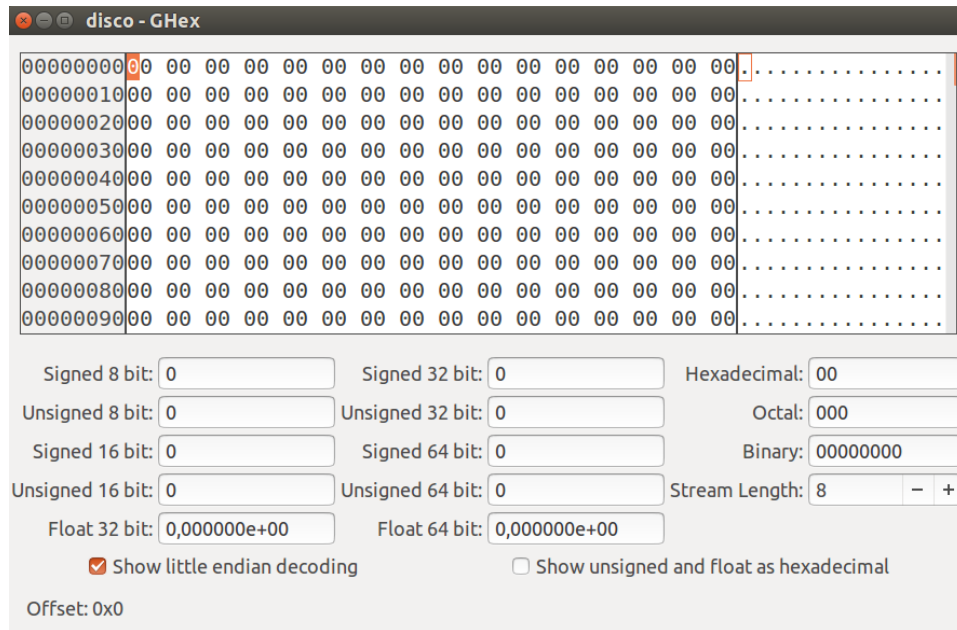


Y luego podéis abrir el fichero con [GHex](#) u [Okteta](#) y comprobar que está todo lleno de 0s:

<sup>4</sup> Si no os aparecen estos permisos utilizad la instrucción `umask(000);` antes de llamar a `open()` en `bmount()`. Ver [máscara de creación del modo de abrir archivos](#)

<sup>5</sup> Aunque es preferible en esta asignatura acostumbrarse a manejar la consola en vez del sistema de ventanas.





## Recursos adicionales:

- [Guía rápida de gcc y gdb](#)
- [C Code Style Guidelines](#)