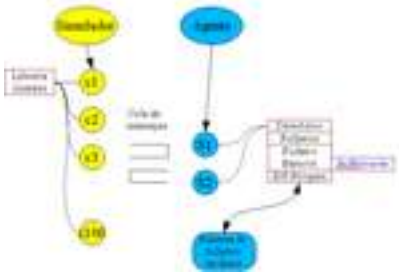

Práctica de ASO
(Ampliación de Sistemas Operativos)

Alejandro Sobrino Beltrán - 43132352-S

3º Ingeniería Técnica en Informática de Sistemas

Enunciado

Se trata de simular un sistema de ficheros basado, en este caso, en inodos. Para ello, se lanzará un agente encargado de arrancar y gestionar los servidores. Posteriormente se ejecutará el simulador, que será el encargado de los clientes.
Los clientes (100 en total) se empezarán a ejecutar uno cada segundo. Cada cliente deberá comunicarse (mediante paso de mensajes) con los servidores para crear un fichero por cliente, denominado "cliente-X.dat" donde X es el *PID* del cliente. Dentro de éste fichero, cada cliente irá escribiendo líneas de log (100 en total) como las siguientes:

```
Inicio log cliente PID
hh:mm::ss Línea número 1
…
hh:mm::ss Línea número n
Fin log cliente PID
```

Para un completo enunciado de la práctica, acudir a http://mnm.uib.es/~gallir/ASO/practica2003.html

Estructura de la práctica

La práctica se ha dividido en los siguientes ficheros de código fuente:

- **agente.c:** código fuente del agente encargado de montar el sistema de ficheros, crear las colas, inicializar el semáforo (al montar el sistema de ficheros) y crear los servidores. Al finalizar la ejecución realiza los pasos oportunos para la eliminación de colas, semáforos así como desmontar el sistema de ficheros.
- **bloques.c:** operaciones de más bajo nivel, a nivel de bloques: montar el sistema de ficheros (*open* del fichero de simulación), desmontarlo (*close*), escribir bloque o leer bloque.
- **cliente.c:** fichero que contiene las operaciones que realizarán los clientes una vez lanzados: creación del fichero cliente-X.dat y escritura en dicho fichero de las líneas de log.
- **cliente_lib.c:** adaptación de las primitivas de acceso para los clientes a mensajes.
- **def.h:** fichero donde se definen las macros que se usarán en toda la práctica.
- **directorio.c:** todas las operaciones (*mount*, *umount*, *create*, *read*, *write*, *unlink*, *ls*, *mkdir*, *stat* y *cat*) se realizan a nivel de nombre de fichero. Se trata del nivel de abstracción mas alto.
- **directorio.h:** definición de la estructura que se usará para asociar el nombre de fichero con su correspondiente inodo.
- **ficheros_basico.c:** se realizan las operaciones básicas sobre el sistema de ficheros: leer/escribir superbloque, leer/escribir inodos, leer/escribir mapa de bits, reserva de inodos/bloques y gestión de la memoria compartida para los servidores. Es importante además asegurar la exclusión mútua a este nivel, ya que en ficheros_basico.c se hayan las operaciones que tratan directamente con los meta-datos (superbloque, inodos y mapa de bits).
- **ficheros_basico.h:** definición de las estructuras de superbloque e inodos.
- **ficheros.c:** las operaciones básicas del sistema de ficheros (*read*, *write*, *truncar* y *stat*) implementadas a nivel de inodos.
- **ficheros.h:** definición de la estructura utilizada por la función *stat* que contendrá información sobre un fichero (o inodo).
- **mensajes.c:** funciones para crear y/o eliminar una cola dado su *id*.
- **mensajes.h:** definición de la estructura utilizada para los mensajes.

- **mi_cat.c:** código fuente del programa para visualizar el contenido de un fichero.
- **mi_ls.c:** código fuente del programa para visualizar el contenido de un directorio.
- **mi_mkfs:** programa encargado de crear e inicializar el sistema de ficheros.
- **mi_stat.c:** código fuente del programa para visualizar información sobre un fichero/directorio.
- **semaforos.c:** funciones para crear, señalizar y eliminar semáforos, así como una implementación de lectores/escritores.
- **servidor.c:** código de las operaciones que deberá realizar el servidor en el sistema de ficheros dependiendo del contenido del mensaje enviado por los clientes.
- **simulador.c:** encargado de ir lanzando a los clientes cada 1 segundo.

Comunicación entre procesos

Durante la simulación de la práctica se deberán ejecutar tres programas:
- **mi_mkfs:** ./mi_mkfs nombre_disco número_de_bloques
- **agente:** ./agente nombre_disco
- **simulador:** ./simulador

mi_mfks se encargará de crear e inicializar el sistema de ficheros (referenciado por el nombre nombre_disco) correctamente. El tamaño del sistema de ficheros vendrá dado por número_de_bloques.

El agente será el encargado de montar el sistema de ficheros y de crear los servidores que estarán a la espera de las peticiones que les hagan los clientes a través de la cola. Una vez realizadas las peticiones y las operaciones, los servidores contestarán a los clientes con otro mensaje, a través de otra cola.

El simulador será el encargado de lanzar a los clientes que se comunicarán con los servidores a través de las dos colas. Realizarán básicamente una operación de creación de fichero, y varias de escritura.

La estructura de los ficheros de código fuente de la práctica están claramente jerarquizados:

```
mi_mkfs -» nombre_disco
simulador -» agente -» servidor -» directorio -» ficheros -» ficheros_basico -»
bloques -» nombre_disco
mi_ls -» nombre_disco
mi_cat -» nombre_disco
mi_stat -» nombre_disco
mi_rm -» nombre_disco
```

Funcionamiento (con ejemplos)

Se deberá empezar por compilar la práctica:

```
jander@hierbabuena:~/uib/aso/practica/src$ make clean
rm -f *.o
jander@hierbabuena:~/uib/aso/practica/src$ make all
gcc -c bloques.c
```

```
gcc -c ficheros_basico.c
gcc -c semaforos.c
gcc -o mi_mkfs mi_mkfs.c bloques.o ficheros_basico.o semaforos.o
gcc -c ficheros.c
gcc -c directorio.c
gcc -c cliente.c
gcc -c cliente_lib.c
gcc -c mensajes.c
gcc -c servidor.c
gcc -o simulador simulador.c bloques.o ficheros_basico.o ficheros.o directorio.o
cliente.o cliente_lib.o mensajes.o semaforos.o servidor.o
gcc -o agente agente.c bloques.o ficheros_basico.o ficheros.o directorio.o
cliente.o cliente_lib.o mensajes.o semaforos.o servidor.o
gcc -o mi_ls mi_ls.c bloques.o ficheros_basico.o ficheros.o directorio.o
semaforos.o
gcc -o mi_cat mi_cat.c bloques.o ficheros_basico.o ficheros.o directorio.o
semaforos.o
gcc -o mi_rm mi_rm.c bloques.o ficheros_basico.o ficheros.o directorio.o
semaforos.o
gcc -o mi_stat mi_stat.c bloques.o ficheros_basico.o ficheros.o directorio.o
semaforos.o
jander@hierbabuena:~/uib/aso/practica/src$
```

A continuación, se procederá a la creación del sistema de ficheros:

```
jander@hierbabuena:~/uib/aso/practica/src$ ./mi_mkfs disco.imagen 10000
(bloques.c|montar_bloques) INFO: Sistema de ficheros montado.
(mi_mkfs.c) tamaño de bloques: 1024
(mi_mkfs.c) número de bloques: 10000
(mi_mkfs.c) bloque del SB: 0
(mi_mkfs.c) bloque inicial del MB: 1
(mi_mkfs.c) bloques para el MB: 2
(mi_mkfs.c) bloque inicial de inodos: 3
(mi_mkfs.c) bloques para inodos: 625
(mi_mkfs.c) bloque inicial de datos: 628
(mi_mkfs.c) primer inodo libre: 1
(bloques.c|desmontar_bloques) INFO: Sistema de ficheros desmontado.
jander@hierbabuena:~/uib/aso/practica/src$
```

Se ejecutará el agente:

```
jander@hierbabuena:~/uib/aso/practica/src$ ./agente disco.imagen
(bloques.c|montar_bloques) INFO: Sistema de ficheros montado.
(agente.c) Creado serdidor [pid: 873].
(agente.c) Creado serdidor [pid: 874].
(agente.c) Servidores creados: 2.

(agente.c) Presione la tecla ENTER cuando haya acabado la simulación.
```

que se quedará a la espera de los clientes, que se lanzarán desde el simulador. Se habrá montado el sistema de ficheros, y creado la memoria compartida (para el superbloque y el mapa de bits), semáforo y colas correspondientes:

```
jander@hierbabuena:~/uib/aso/practica/src$ ipcs -a

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
```

```
0x00000000 1474564    jander    666        36         3
0x00000000 1507333    jander    666        2          3

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00000000 229379     jander    600        1

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00000032 131072     jander     600        0            0
0x00000033 163841     jander     600        0            0
jander@hierbabuena:~/uib/aso/practica/src$
```

Posteriormente, sólo queda lanzar el simulador:

```
jander@hierbabuena:~/uib/aso/practica/src$ ./simulador
(cliente.c) INFO: creado fichero /cliente-885.dat por cliente 885.
(cliente.c) INFO: creado fichero /cliente-886.dat por cliente 886.
(cliente.c) INFO: creado fichero /cliente-887.dat por cliente 887.
(cliente.c) INFO: creado fichero /cliente-888.dat por cliente 888.
(cliente.c) INFO: creado fichero /cliente-889.dat por cliente 889.
(cliente.c) INFO: creado fichero /cliente-890.dat por cliente 890.
(cliente.c) INFO: creado fichero /cliente-891.dat por cliente 891.
(cliente.c) INFO: creado fichero /cliente-892.dat por cliente 892.
(cliente.c) INFO: creado fichero /cliente-893.dat por cliente 893.
(cliente.c) INFO: creado fichero /cliente-894.dat por cliente 894.
(cliente.c) INFO: creado fichero /cliente-895.dat por cliente 895.
(cliente.c) INFO: creado fichero /cliente-896.dat por cliente 896.
(cliente.c) INFO: creado fichero /cliente-897.dat por cliente 897.
      .....
(cliente.c) INFO: creado fichero /cliente-981.dat por cliente 981.
(cliente.c) INFO: creado fichero /cliente-982.dat por cliente 982.
(cliente.c) INFO: creado fichero /cliente-983.dat por cliente 983.
(cliente.c) INFO: creado fichero /cliente-984.dat por cliente 984.
(cliente.c) INFO: creado fichero /cliente-985.dat por cliente 985.
(cliente.c) INFO: creado fichero /cliente-986.dat por cliente 986.
(cliente.c) INFO: creado fichero /cliente-987.dat por cliente 987.
(simulador.c) Clientes creados: 100.
(simulador.c) Acabados: 1.
(simulador.c) Acabados: 2.
(simulador.c) Acabados: 3.
(simulador.c) Acabados: 4.
(simulador.c) Acabados: 5.
(simulador.c) Acabados: 6.
(simulador.c) Acabados: 7.
(simulador.c) Acabados: 8.
      .....
(simulador.c) Acabados: 94.
(simulador.c) Acabados: 96.
(simulador.c) Acabados: 97.
(simulador.c) Acabados: 98.
(simulador.c) Acabados: 99.
(simulador.c) Acabados: 100.
jander@hierbabuena:~/uib/aso/practica/src$
```

Una salida de un ps aux durante la ejecución de la práctica:

```
jander@hierbabuena:~$ ps aux
```

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.2  0.1   1528   528 ?        S    07:41   0:04 init [2]
      .....
jander     878  0.0  0.0   1416   368 pts/1    S+   08:12   0:00 ./agente
disco.imagen
jander     879  0.2  0.0   1544   404 pts/1    S+   08:12   0:00 ./agente
disco.imagen
jander     880  0.2  0.0   1544   400 pts/1    S+   08:12   0:00 ./agente
disco.imagen
jander     881  0.0  0.3   3124  1744 pts/3    Ss   08:12   0:00 /bin/bash
jander     884  0.0  0.0   1400   272 pts/2    S+   08:13   0:00 ./simulador
jander     885  0.0  0.0   1536   440 pts/2    S+   08:13   0:00 ./simulador
jander     886  0.0  0.0   1536   440 pts/2    S+   08:13   0:00 ./simulador
jander     887  0.0  0.0   1536   440 pts/2    S+   08:13   0:00 ./simulador
jander     888  0.0  0.0   1536   440 pts/2    S+   08:13   0:00 ./simulador
jander     889  0.0  0.0   1536   440 pts/2    S+   08:13   0:00 ./simulador
jander     890  0.0  0.0   1536   440 pts/2    S+   08:13   0:00 ./simulador
jander     891  0.0  0.0   1536   440 pts/2    S+   08:13   0:00 ./simulador
jander     892  0.0  0.0   1536   440 pts/2    S+   08:13   0:00 ./simulador
      .....
jander     928  0.0  0.0   1536   440 pts/2    S+   08:13   0:00 ./simulador
jander     929  0.0  0.0   1536   440 pts/2    S+   08:13   0:00 ./simulador
jander     930  0.0  0.0   1536   440 pts/2    S+   08:13   0:00 ./simulador
jander     931  0.0  0.0   1536   440 pts/2    S+   08:13   0:00 ./simulador
jander     932  0.0  0.1   2524   840 pts/3    R+   08:13   0:00 ps aux
jander@hierbabuena:~$ ps aux |grep simulador | wc -l
67
jander@hierbabuena:~$ ps aux |grep simulador | wc -l
47
jander@hierbabuena:~$ ps aux |grep simulador | wc -l
36
```

Durante la ejecución del simulador, se crean los 100 clientes (con un segundo entre la creación de cada uno de ellos). Cada uno de los clientes, generará un mensaje con el código de operación correspondiente a la creacicón de fichero. El nombre del fichero será cliente-X.dat donde X es el *PID* del cliente.
Seguidamente, los clientes irán generando, de la misma manera, las operaciones de escritura sobre los ficheros previamente creados. Las líneas de log que serán escritas en el fichero seguirán la forma señalada en el enunciado de la práctica. Una vez finalizado todo el proceso, el simulador acaba.
Al otro lado de las colas se hayan los servidores que al recibir el mensaje por parte de un cliente lo primero que hacen es comprobar el tipo de operación a realizar. Se cojerán los datos necesarios del mensaje y se realizará la operación indicada en el mensaje. Así sucesivamente.
Una vez haya finalizado la simulación, bastará con presionar la tecla *ENTER* en el agente, para finalizar con la simulación de la práctica.

```
(agente.c) Matando servidor [pid: 879].
(agente.c) Matando servidor [pid: 880].
(agente.c) Acabados: 2.
(bloques.c|desmontar_bloques) INFO: Sistema de ficheros desmontado.
jander@hierbabuena:~/uib/aso/practica/src$
```

Las colas serán eliminadas al igual que los semáforos y la memoria compartida. Además, se desmontará el sistema de ficheros, dando así por finalizada la simulación.

Posteriormente, se podrán ejecutar alguno de los demás binarios:

mi_ls

```
jander@hierbabuena:~/uib/aso/practica/src$ ./mi_ls disco.imagen /
(bloques.c|montar_bloques) INFO: Sistema de ficheros montado.
(mi_ls.c) Número de ficheros: 100.

cliente-885.dat
cliente-886.dat
cliente-887.dat
cliente-888.dat
cliente-889.dat
cliente-890.dat
cliente-891.dat
cliente-892.dat
cliente-893.dat
      .....
cliente-984.dat
cliente-985.dat
cliente-986.dat
cliente-987.dat

(bloques.c|desmontar_bloques) INFO: Sistema de ficheros desmontado.
jander@hierbabuena:~/uib/aso/practica/src$
```

mi_stat

```
jander@hierbabuena:~/uib/aso/practica/src$ ./mi_stat disco.imagen /cliente-987.dat
(bloques.c|montar_bloques) INFO: Sistema de ficheros montado.
Inodo: 100
Fecha de creación: Thu Sep  2 08:13:03 2004
Fecha de modificación: Thu Sep  2 08:13:31 2004
Fecha de último acceso: Thu Sep  2 08:13:31 2004
Tamaño en bytes: 2676.
(bloques.c|desmontar_bloques) INFO: Sistema de ficheros desmontado.
jander@hierbabuena:~/uib/aso/practica/src$
```

mi_cat

```
jander@hierbabuena:~/uib/aso/practica/src$ ./mi_cat disco.imagen /cliente-987.dat
(bloques.c|montar_bloques) INFO: Sistema de ficheros montado.
Inicio del log [cliente: 1156]
08:13::18 Linea número 0
08:13::18 Linea número 1
08:13::19 Linea número 2
08:13::19 Linea número 3
08:13::19 Linea número 4
08:13::19 Linea número 5
08:13::19 Linea número 6
08:13::19 Linea número 7
08:13::19 Linea número 8
08:13::19 Linea número 9
08:13::19 Linea número 10
      .....
08:13::30 Linea número 95
08:13::30 Linea número 96
08:13::30 Linea número 97
08:13::31 Linea número 98
08:13::31 Linea número 99
08:13::31 Linea número 100
Fin del log [cliente: 1172]
(bloques.c|desmontar_bloques) INFO: Sistema de ficheros desmontado.
jander@hierbabuena:~/uib/aso/practica/src$
```

Consideraciones

Algunas de las consideraciones a tener en cuenta:

1. El tamaño de bloques es variable (bastará con cambiar el valor a TAM_BLOQUE en def.h) y recompilar la práctica.
2. Existe la posibilidad de *DEBUG* en la que todas las funciones imprimirán el máximo de información posible.
3. Cualquier error o fallo producido por alguna función será reproducido por pantalla con el formato: `(nombre_fichero|nombre_función) ERROR: mensaje de error.`
4. La exclusión mútua se ha hecho a nivel de ficheros_basico en las funciones que hacen uso de meta-información a través del uso de un *mutex*.
5. El agente y el simulador están basados en la práctica de SO (Sistemas Operativos) de 2002-2003 @ UIB: http://bulma.net/~jander/uib/so/

Código fuente

Makefile

```
# General
all:              mi_mkfs simulador agente mi_ls mi_cat mi_rm mi_stat
clean:
                  rm -f *.o

# Binarios
mi_mkfs:          mi_mkfs.c bloques.o ficheros_basico.o semaforos.o
                  gcc -o mi_mkfs mi_mkfs.c bloques.o ficheros_basico.o semaforos.o
simulador:        simulador.c bloques.o ficheros_basico.o ficheros.o directorio.o
cliente.o cliente_lib.o mensajes.o semaforos.o servidor.o
                  gcc -o simulador simulador.c bloques.o ficheros_basico.o
ficheros.o directorio.o cliente.o cliente_lib.o mensajes.o semaforos.o servidor.o
agente:           agente.c bloques.o ficheros_basico.o ficheros.o directorio.o
cliente.o cliente_lib.o mensajes.o semaforos.o servidor.o
                  gcc -o agente agente.c bloques.o ficheros_basico.o ficheros.o
directorio.o cliente.o cliente_lib.o mensajes.o semaforos.o servidor.o
mi_ls:                  mi_ls.c bloques.o ficheros_basico.o ficheros.o directorio.o
semaforos.o
                  gcc -o mi_ls mi_ls.c bloques.o ficheros_basico.o ficheros.o
directorio.o semaforos.o
mi_cat:                 mi_cat.c bloques.o ficheros_basico.o ficheros.o
directorio.o semaforos.o
                  gcc -o mi_cat mi_cat.c bloques.o ficheros_basico.o ficheros.o
directorio.o semaforos.o
mi_rm:                  mi_rm.c bloques.o ficheros_basico.o ficheros.o directorio.o
semaforos.o
                  gcc -o mi_rm mi_rm.c bloques.o ficheros_basico.o ficheros.o
directorio.o semaforos.o
mi_stat:          mi_stat.c bloques.o ficheros_basico.o ficheros.o directorio.o
semaforos.o
                  gcc -o mi_stat mi_stat.c bloques.o ficheros_basico.o ficheros.o
directorio.o semaforos.o

# Pruebas
```

```
pruebas1:         pruebas1.c bloques.o ficheros_basico.o
                  gcc -o pruebas1 pruebas1.c ficheros_basico.o bloques.o
pruebas2:         pruebas2.c bloques.o ficheros_basico.o
                  gcc -o pruebas2 pruebas2.c ficheros_basico.o bloques.o
pruebas3:         pruebas3.c bloques.o ficheros_basico.o ficheros.o
                  gcc -o pruebas3 pruebas3.c ficheros.o ficheros_basico.o bloques.o
pruebas4:         pruebas4.c bloques.o ficheros_basico.o ficheros.o directorio.o
                  gcc -o pruebas4 pruebas4.c bloques.o ficheros_basico.o ficheros.o
directorio.o

# Objetos
bloques.o:              bloques.c bloques.h
                        gcc -c bloques.c
ficheros_basico.o:      ficheros_basico.c ficheros_basico.h
                        gcc -c ficheros_basico.c
ficheros.o:             ficheros.c ficheros.h
                        gcc -c ficheros.c
directorio.o:           directorio.c directorio.h
                        gcc -c directorio.c
cliente.o:              cliente.c cliente.h
                        gcc -c cliente.c
cliente_lib.o:          cliente_lib.c cliente_lib.h
                        gcc -c cliente_lib.c
mensajes.o:             mensajes.c mensajes.h
                        gcc -c mensajes.c
semaforos.o:            semaforos.c
                        gcc -c semaforos.c
servidor.o:             servidor.c servidor.h
                        gcc -c servidor.c
```

agente.c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/resource.h>
#include "def.h"
#include "mensajes.h"
#include "servidor.h"
#include "directorio.h"

int n_acabados;
int pid_servidores[N_SERVIDORES];

void enterrador ()
{
      int n_enterrados = 0;
      while (wait3(NULL,WNOHANG,NULL) > 0) {
            n_enterrados++;
            n_acabados++;
      }
      if (n_enterrados > 0) {
            printf("(agente.c) Acabados: %d.\n",n_acabados);
      }
}

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

```
{
      setlinebuf(stdout);
      // comprobación de argumentos
      if (argc != 2) {
            printf("(agente.c) usar %s nombre_sistema_ficheros.\n",argv[0]);
            return -1;
      }
      /* montamos el sistema de ficheros
       * creamos e inicializamos la memoria compartida
       * creamos e inicializamos el mutex
       */
      if (montar(argv[1]) == -1) {
            return -1;
      }
      int cola_r,cola_s;
      // creamos las colas
      cola_s = crear_cola(50);
      cola_r = crear_cola(51);
      n_acabados = 0;
      int i;
      int n_creados = 0;
      // creamos los servidores
      for (i=0;i<N_SERVIDORES;i++) {
            if ((pid_servidores[i]=fork()) == 0) {
                  printf("(agente.c) Creado serdidor [pid: %d].\n",getpid());
                  servidor(cola_s,cola_r);
                  exit(0);
            } else if (pid_servidores[i] > 0) {
                  n_creados++;
                  usleep(100000);
            } else {
                  printf("(agente.c) ERROR: Imposible crear el servidor.\n");
                  break;
            }
      }
      printf("(agente.c) Servidores creados: %d.\n",n_creados);
      printf("\n(agente.c) Presione la tecla ENTER cuando haya acabado la
simulación.\n\n");
      // esperamos a que presionen la tecla ENTER
      getchar();
      for (i=0;i<n_creados;i++) {
            printf("(agente.c) Matando servidor [pid: %d].\n",pid_servidores[i]);
            kill(pid_servidores[i],SIGTERM);
      }
      while (n_acabados < n_creados) {
            enterrador();
      }
      /* eliminamos las colas
       * desmontamos el sistema de fichero
       * eliminamos memoria compartida
       * eliminamos semáforo mutex
       */
      eliminar_cola(cola_r);
      eliminar_cola(cola_s);
      desmontar();
}
```

bloques.c

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

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include "def.h"

static int dsf;

// Montamos el sistema de ficheros (open)
int montar_bloques (char *nombre)
{
      if ((dsf = open(nombre,O_CREAT|O_RDWR,S_IRUSR|S_IWUSR)) == -1) {
            printf("(bloques.c|montar_bloques) ERROR: problema al montar el sistema
de ficheros.\n");
            return -1;
      }
      printf("(bloques.c|montar_bloques) INFO: Sistema de ficheros montado.\n");
      return dsf;
}

// desmontamos el sistema de ficheros (close)
int desmontar_bloques ()
{
      if (close(dsf) == -1) {
            printf("(bloques.c|desmontar_bloques) ERROR: Problema al desmontar el
sistema de ficheros.\n");
            return -1;
      }
      printf("(bloques.c|desmontar_bloques) INFO: Sistema de ficheros
desmontado.\n");
      return 0;
}

// leemos un bloque determinado
int leer_bloque (int n_bloque, char *buffer)
{
      if (n_bloque < 0) {
            return -1;
      }
      int offset = n_bloque * TAM_BLOQUE;
      if (lseek(dsf,offset,SEEK_SET) == (offset-1)) {
            printf("(bloques.c|leer_bloque) ERROR: Problema con el lseek [offset: %
d].\n",offset);
            return -1;
      }
      if (read(dsf,buffer,TAM_BLOQUE) == -1) {
            printf("(bloques.c|leer_bloque) ERROR: problema con la lectura del
bloque [n_bloque: %d].\n",n_bloque);
            return -1;
      }
      if (DEBUG == 1) {
            printf("(bloques.c|leer_bloque) DEBUG: leido bloque %d offset %
d.\n",n_bloque,offset);
      }
      return 0;
}

// escribimos un bloque determinado
int escribir_bloque (int n_bloque, char *buffer)
{
      if (n_bloque < 0) {
```

```
            return -1;
      }
      int offset = n_bloque * TAM_BLOQUE;
      if (lseek(dsf,offset,SEEK_SET) == (offset-1)) {
            printf("(bloques.c|escribir_bloque) ERROR: Problema con el lseek
[offset: %d].\n",offset);
            return -1;
      }
      int tmp;
      if ((tmp = write(dsf,buffer,TAM_BLOQUE)) == -1) {
            printf("(bloques.c|escribir_bloque) ERROR: Problema con la lectura del
bloque [n_bloque: %d].\n",n_bloque);
            return -1;
      }
      if (tmp != TAM_BLOQUE) {
            printf("(bloques.c|escribir_bloque) ERROR: se han escrito %d bytes en
vez de %d.\n",tmp,TAM_BLOQUE);
      }
      if (DEBUG == 1) {
            printf("(bloques.c|escribir_bloque) DEBUG: escrito bloque %d offset %
d.\n",n_bloque,offset);
      }
      return 0;
}
```

bloques.h

```
int montar_bloques (char *);
int desmontar_bloques ();
int leer_bloque (int, char *);
int escribir_bloque (int, char *);
```

cliente.c

```
#include <time.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "cliente.h"
#include "directorio.h"
#include "cliente_lib.h"

/* función encargada de crear el fichero cliente-X.dat (X pid del cliente) y
 * realizar las escrituras de las líneas de log a través de las funciones de la
 * librería de clientes cliente.lib.
 */
void cliente (int cola_s, int cola_r)
{
      char nombre[MAX_NOMBRE];
      char buffer[MAX_LINEA];
      memset(buffer,'\0',MAX_NOMBRE);
      memset(buffer,'\0',MAX_LINEA);
      sprintf(nombre,"/cliente-%d.dat",getpid());
      mi_create_cliente(nombre,cola_s,cola_r);
      printf("(cliente.c) INFO: creado fichero %s por cliente %d.\n",nombre,getpid
());
      sprintf(buffer,"Inicio del log [cliente: %d]\n",getpid());
      mi_write_cliente(nombre,0,strlen(buffer),buffer,cola_s,cola_r);
      int posicion = strlen(buffer);
      int i,tmp;
      time_t t;
```

```
      struct tm *p_t;
      for (i=0;i<MAX_LOG+1;i++) {
            t = time(NULL);
            p_t = localtime(&t);
            memset(buffer,'\0',MAX_LINEA);
            tmp = strftime(buffer,MAX_LINEA,"%R::%S ",p_t);
            sprintf(buffer+tmp,"Linea número %d\n",i);
            mi_write_cliente(nombre,posicion,strlen(buffer),buffer,cola_s,cola_r);
            posicion = posicion + strlen(buffer);
            usleep(100000);
      }
      memset(buffer,'\0',MAX_LINEA);
      sprintf(buffer,"Fin del log [cliente: %d]\n",getpid());
      mi_write_cliente(nombre,posicion,strlen(buffer),buffer,cola_s,cola_r);
}
```

cliente.h

```
void cliente (int, int);
```

cliente_lib.c

```
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>
#include "def.h"
#include "mensajes.h"
#include "ficheros.h"
#include "cliente_lib.h"

int mi_write_cliente (char *path, int posicion, int tamano, char *buffer, int
cola_s, int cola_r)
{
      mensaje msg1,msg2;
      strcpy(msg1.contenido.nombre,path);
      msg1.contenido.tipo = 1;
      msg1.pid = getpid();
      int i;
      int tmp = tamano/TAM_BLOQUE;
      int resto = 0;
      if (tamano % TAM_BLOQUE != 0) {
            tmp++;
            resto = tamano % TAM_BLOQUE;
      } else {
            resto = TAM_BLOQUE;
      }
      // vamos enviando los mensajes a los servidores
      for (i=0;i<tmp;i++) {
            if (i==(tmp-1)) {
                  memcpy(&msg1.contenido.info,&buffer[i*TAM_BLOQUE],resto);
                  msg1.contenido.tamano = resto;
                  msg1.contenido.posicion = posicion + (i*TAM_BLOQUE);
            } else {
                  memcpy(&msg1.contenido.info,&buffer[i*TAM_BLOQUE],TAM_BLOQUE);
                  msg1.contenido.tamano = TAM_BLOQUE;
                  msg1.contenido.posicion = posicion + (i*TAM_BLOQUE);
            }
            if (msgsnd(cola_s,&msg1,sizeof(c_mensaje),0) < 0) {
                  printf("(cliente_lib.c) ERROR: imposible enviar el mensaje [cola:
%d].\n",cola_s);
```

```
            }
            if (msgrcv(cola_r,&msg2,sizeof(c_mensaje),getpid(),0) < 0) {
                  printf("(cliente_lib.c) ERROR: imposible recibir el mensaje
[cola: %d].\n",cola_r);
            }
      }
      return 0;
}

int mi_read_cliente (char *path, int posicion, int tamano, char *buffer, int
cola_s, int cola_r)
{
      mensaje msg1,msg2;
      strcpy(msg1.contenido.nombre,path);
      msg1.contenido.tipo = 5;
      msg1.pid = getpid();
      int tmp = tamano/TAM_BLOQUE;
      int resto = 0;
      if (tamano % TAM_BLOQUE != 0) {
            tmp++;
            resto = tamano%TAM_BLOQUE;
      } else {
            resto = TAM_BLOQUE;
      }
      int i;
      // vamos enviando los mensajes a los servidores
      for (i=0;i<tmp;i++) {
            if (i==(tmp-1)) {
                  memcpy(&msg1.contenido.info,&buffer[i*TAM_BLOQUE],resto);
                  msg1.contenido.tamano = resto;
                  msg1.contenido.posicion = posicion + (i*TAM_BLOQUE);
            } else {
                  memcpy(&msg1.contenido.info,&buffer[i*TAM_BLOQUE],TAM_BLOQUE);
                  msg1.contenido.tamano = TAM_BLOQUE;
                  msg1.contenido.posicion = posicion + (i*TAM_BLOQUE);
            }
            if (msgsnd(cola_s,&msg1,sizeof(c_mensaje),0) < 0) {
                  printf("(cliente_lib.c) ERROR: imposible enviar el mensaje [cola:
%d].\n",cola_s);
            }
            if (msgrcv(cola_r,&msg2,sizeof(c_mensaje),getpid(),0) < 0) {
                  printf("(cliente_lib.c) ERROR: imposible recibir el mensaje
[cola: %d].\n",cola_r);
            }
      }
      return 0;
}

int mi_create_cliente (char *path, int cola_s, int cola_r)
{
      mensaje msg1,msg2;
      strcpy(msg1.contenido.nombre,path);
      msg1.contenido.tipo = 0;
      msg1.contenido.tamano = 0;
      msg1.contenido.posicion = 0;
      msg1.pid = getpid();
      if (msgsnd(cola_s,&msg1,sizeof(c_mensaje),0) < 0) {
            printf("(cliente_lib.c) ERROR: imposible enviar el mensaje [cola: %d].
\n",cola_s);
      }
```

```
      if (msgrcv(cola_r,&msg2,sizeof(c_mensaje),getpid(),0) < 0) {
            printf("(cliente_lib.c) ERROR: imposible recibir el mensaje [cola: %d].
\n",cola_r);
      }
      return 0;
}

int mi_dir_cliente (char *path, char *buffer, int cola_s, int cola_r)
{
      mensaje msg1,msg2;
      strcpy(msg1.contenido.nombre,path);
      strcpy(msg1.contenido.info,buffer);
      msg1.contenido.tipo = 2;
      msg1.contenido.tamano = 0;
      msg1.contenido.posicion = 0;
      msg1.pid = getpid();
      if (msgsnd(cola_s,&msg1,sizeof(c_mensaje),0) < 0) {
            printf("(cliente_lib.c) ERROR: imposible enviar el mensaje [cola: %d].
\n",cola_s);
      }
      if (msgrcv(cola_r,&msg2,sizeof(c_mensaje),getpid(),0) < 0) {
            printf("(cliente_lib.c) ERROR: imposible recibir el mensaje [cola: %d].
\n",cola_r);
      }
      // imprimimos el resultado
      int i = 0;
      printf("(cliente_lib.c) INFO: listado del directorio %s con %d
ficheros.\n",path,msg2.contenido.tamano);
      while (msg2.contenido.info[i] != '\0') {
            if (msg2.contenido.info[i] == ':') {
                  printf("\n");
            } else {
                  printf("%c",msg2.contenido.info[i]);
            }
            i++;
      }
      // lo copiamos al buffer
      strcpy(buffer,info);
      printf("\n");
      return 0;
}

int mi_rm_cliente (char *path, int cola_s, int cola_r)
{
      mensaje msg1,msg2;
      strcpy(msg1.contenido.nombre,path);
      msg1.contenido.tipo = 3;
      msg1.contenido.tamano = 0;
      msg1.contenido.posicion = 0;
      msg1.pid = getpid();
      if (msgsnd(cola_s,&msg1,sizeof(c_mensaje),0) < 0) {
            printf("(cliente_lib.c) ERROR: imposible enviar el mensaje [cola: %d].
\n",cola_s);
      }
      if (msgrcv(cola_r,&msg2,sizeof(c_mensaje),getpid(),0) < 0) {
            printf("(cliente_lib.c) ERROR: imposible recibir el mensaje [cola: %d].
\n",cola_r);
      }
      return 0;
}
```

```
int mi_stat_cliente (char *path, int cola_s, int cola_r)
{
      mensaje msg1,msg2;
      estat st;
      strcpy(msg1.contenido.nombre,path);
      msg1.contenido.tipo = 4;
      msg1.contenido.tamano = 0;
      msg1.contenido.posicion = 0;
      msg1.pid = getpid();
      if (msgsnd(cola_s,&msg1,sizeof(c_mensaje),0) < 0) {
            printf("(cliente_lib.c) ERROR: imposible enviar el mensaje [cola: %d].
\n",cola_s);
      }
      if (msgrcv(cola_r,&msg2,sizeof(c_mensaje),getpid(),0) < 0) {
            printf("(cliente_lib.c) ERROR: imposible recibir el mensaje [cola: %d].
\n",cola_r);
      }
      memcpy(&st,msg2.contenido.info,sizeof(estat));
      // imprimimos el resultado
      printf("(cliente_lib.c) INFO: archivo %s.\n",path);
      printf("(cliente_lib.c) INFO: inodo %d con longitud %
d.\n",st.n_inodo,st.longitud);
      printf("(cliente_lib.c) INFO: fecha creación %s - fecha modificación %s -
fecha de último acceso %s.\n",ctime(&st.f_creacion),ctime(&st.f_modificacion),ctime
(&st.f_ultimoacceso));
      return 0;
}
```

cliente_lib.h

```
int mi_read_cliente (char *, int, int, char *, int, int);
int mi_write_cliente (char *, int, int, char *, int, int);
int mi_create_cliente (char *, int, int);
int mi_stat_cliente (char *, int, int);
int mi_rm_cliente (char *, int, int);
int mi_dir_cliente (char *, char *, int, int);
```

def.h

```
/*
 * DEBUG [def: 0]
 * 0: desactivado.
 * 1: activado.
 */
#define DEBUG 0
/*
 * TAM_BLOQUE [def: 1024]
 * tamaño de bloque del sistema de ficheros.
 */
#define TAM_BLOQUE 1024
/*
 * MAX_PTROS [def: 10]
 * número máximo de punteros por inodo.
 */
#define MAX_PTROS 10
/*
 * MAX_NOMBRE [def: 28]
 * máximo número de caracteres para los nombres de fichero/directorio.
 */
#define MAX_NOMBRE 28
```

```
/*
 * N_SERVIDORES [def: 2]
 * número de servidores.
 */
#define N_SERVIDORES 2
/*
 * N_CLIENTES [def: 100]
 * número de clientes.
 */
#define N_CLIENTES 100
/*
 * MAX_LOG [def: 100]
 *
 */
#define MAX_LOG 100
/*
 * MAX_LINEA [def: 50]
 *
 */
#define MAX_LINEA 50
```

directorio.c

```
#include <stdlib.h>
#include <string.h>
#include "def.h"
#include "ficheros_basico.h"
#include "ficheros.h"
#include "directorio.h"
#include "semaforos.h"

int montar (char *nombre)
{
      return montar_basico(nombre);
}

int desmontar (void)
{
      return desmontar_basico();
}

// busca el inodo del fichero nombre a partir de su directorio con inodo in
int buscar_fichero (char *nombre, inodo in, int *n_inodo)
{
      entrada e;
      int posicion = 0;
      char buffer[sizeof(entrada)];
      if (in.longitud == 0) {
            printf("(directorio.c|buscar_fichero) INFO: longitud del inodo %d es
0.\n",in.n_inodo);
            return -1;
      }
      int i;
      int leido = 0;
      while (posicion < in.longitud) {
            if ((leido = mi_read_ficheros(*n_inodo,posicion,sizeof(entrada),
buffer)) < 0) {
                  printf("(directorio.c|buscar_fichero) ERROR: lectura incompleta
[n_inodo: %d  posicion: %d  tamaño: %d  leido: %d].\n",*n_inodo,posicion,sizeof
(entrada),leido);
```

```
            }
            memcpy(&e,buffer,sizeof(entrada));
            i = 0;
            while (nombre[i] == e.nombre[i]) {
                  if ((nombre[i] == '\0') && (i>0)) {
                        *n_inodo = e.n_inodo;
                        return 0;
                  }
                  i++;
            }
            posicion = posicion + sizeof(entrada);
      }
      return -1;
}

/* busca el inodo del fichero recursivamente dentro del directorio (normalmente
 * /, inodo 0)
 */
int buscar_directorio (char *nombre, inodo in, int *n_inodo, int control, char
*fichero)
{
      char nombre_tmp[MAX_NOMBRE];
      int i,j;
      for (i=0;i<MAX_NOMBRE-1;i++) {
            nombre_tmp[i] = '\0';
      }
      if (nombre[control] != '/') {
            printf("(directorio.c|buscar_directorio) ERROR: el caracter %d del
nombre debe ser '/'.\n",control);
            return -1;
      }
      i = 0;
      j = control+1;
      while ((nombre[j] != '\0') && (nombre[j] != '/')) {
            nombre_tmp[i] = nombre[j];
            i++; j++;
      }
      nombre_tmp[i] = '\0';
      if (nombre[j] == '\0') {
            strcpy(fichero,nombre_tmp);
      }
      if (buscar_fichero(nombre_tmp,in,n_inodo) == 0) {
            if (nombre[j] == '/') {
                  if (leer_inodo(*n_inodo,&in) == -1) {
                        printf("(directorio.c|buscar_directorio) ERROR: imposible
leer el inodo %d.\n",*n_inodo);
                        return -1;
                  }
                  if (buscar_directorio(nombre,in,n_inodo,j,fichero) == 0) {
                        return 0;
                  } else {
                        return -1;
                  }
            }
      }
      return 0;
}

int mi_create_directorio (char *nombre)
{
```

```
      entrada e;
      int n_inodo = 0;
      char nombre_tmp[MAX_NOMBRE];
      char *cadena;
      inodo in;
      int escrito = 0;
      cadena = (char *)(malloc((strlen(nombre)*sizeof(char))+1));
      char buffer[sizeof(entrada)];
      strcpy(cadena,nombre);
      if (leer_inodo(0,&in) == -1) {
            printf("(directorio.c|mi_create_directorio) ERROR: imposible leer inodo
0.\n");
            return -1;
      }
      if (buscar_directorio(cadena,in,&n_inodo,0,nombre_tmp) == -1) {
            return -1;
      }
      // el archivo ya existe?
      if (in.n_inodo != n_inodo) {
            printf("(directorio.c|mi_create_directorio) ERROR: el archivo ya
existe.\n");
            return -1;
      }
      // añadimos la entrada al directorio
      strcpy(e.nombre,nombre_tmp);
      e.n_inodo = reservar_inodo();
      memcpy(buffer,&e,sizeof(entrada));
      if ((escrito = mi_write_ficheros(n_inodo,in.longitud,sizeof(entrada),buffer))
< sizeof(entrada)) {
            printf("(directorio.c|mi_create_directorio) ERROR: escritura incompleta
[n_inodo: %d  posicion: %d  tamaño: %d  escrito: %d].\n",n_inodo,in.longitud,sizeof
(entrada),escrito);
      }
      free(cadena);
      return 0;
}

int mi_read_directorio (char *nombre, int posicion, int tamano, char *buffer)
{
      inodo in;
      int n_inodo = 0;
      char nombre_tmp[MAX_NOMBRE];
      int leido = 0;
      if (leer_inodo(0,&in) == -1) {
            printf("(directorio.c|mi_read_directorio) ERROR: imposible leer inodo
0.\n");
            return -1;
      }
      // buscamos el inodo del fichero
      if (buscar_directorio(nombre,in,&n_inodo,0,nombre_tmp) == -1) {
            printf("(directorio.c|mi_read_directorio) ERROR: imposible encontrar
directorio.\n");
            return -1;
      }
      if ((leido = mi_read_ficheros(n_inodo,posicion,tamano,buffer)) < tamano) {
            printf("(directorio.c|mi_read_directorio) ERROR: lectura incompleta
[n_inodo: %d  posicion: %d  tamaño: %d  leido: %d].
\n",n_inodo,posicion,tamano,leido);
            return leido;
      }
```

```
	return leido;
}

int mi_unlink_directorio (char *nombre)
{
	entrada e;
	int n_inodo = 0;
	inodo in1,in2;
	char nombre_tmp[MAX_NOMBRE];
	char buffer[sizeof(entrada)];
	int leido,escrito;
	leido = escrito = 0;
	if (leer_inodo(n_inodo,&in1) == -1) {
		printf("(directorio.c|mi_unlink_directorio) ERROR: imposible leer inodo
%d.\n",n_inodo);
		return -1;
	}
	// buscamos el inodo del fichero
	if (buscar_directorio(nombre,in1,&n_inodo,0,nombre_tmp) < 0) {
		printf("(directorio.c|mi_unlink_directorio) ERROR: imposible encontrar
directorio.\n");
		return -1;
	}
	// el archivo no existe?
	if (in1.n_inodo == n_inodo) {
		printf("(directorio.c|mi_unlink_directorio) ERROR: el archivo no
existe.\n");
		return -1;
	}
	if (leer_inodo(n_inodo,&in2) == -1) {
		printf("(directorio.c|mi_unlink_directorio) ERROR: imposible leer inodo
%d.\n",n_inodo);
		return -1;
	}
	if ((in1.longitud > 0) && (in2.tipo == 2)) {
		printf("(directorio.c|mi_unlink_directorio) ERROR: el directorio
contiene ficheros.\n");
		return -1;
	}
	int i;
	int posicion = 0;
	while (posicion < in1.longitud) {
		/* leemos la última entrada
		 * si es la última entrada, la truncamos. Sino, copiamos la última a
		 * la encontrada (sobreescribiéndola) y borramos la última.
		 */
		if ((leido = mi_read_ficheros(in1.n_inodo,(in1.longitud-sizeof
(entrada)),sizeof(entrada),buffer)) < sizeof(entrada)) {
			printf("(directorio.c|mi_unlink_directorio) ERROR: lectura
incompleta [n_inodo: %d  posicion: %d  tamaño: %d  leido: %d].\n",in1.n_inodo,
(in1.longitud-sizeof(entrada)),sizeof(entrada),leido);
		}
		memcpy(&e,buffer,sizeof(entrada));
		i = 0;
		while (nombre_tmp[i] == e.nombre[i]) {
			if ((nombre_tmp[i] == '\0') && (i>0)) {
				if ((leido = mi_read_ficheros(in1.n_inodo,(in1.longitud-
sizeof(entrada)),sizeof(entrada),buffer)) < sizeof(entrada)) {
					printf("(directorio.c|mi_unlink_directorio) ERROR:
lectura incompleta [n_inodo: %d  posicion: %d  tamaño: %d  leido: %d].
```

```
\n",in1.n_inodo,(in1.longitud-sizeof(entrada)),sizeof(entrada),leido);
                        }
                        if ((escrito = mi_write_ficheros
(in1.n_inodo,posicion,sizeof(entrada),buffer)) < sizeof(entrada)) {
                              printf("(directorio.c|mi_unlink_directorio) ERROR:
lectura incompleta [n_inodo: %d  posicion: %d tamaño: %d  escrito: %d].
\n",in1.n_inodo,posicion,sizeof(entrada),escrito);
                        }
                        if (mi_truncar_ficheros(in1.n_inodo,(in1.longitud-sizeof
(entrada))) == -1) {
                              printf("(directorio.c|mi_unlink_directorio) ERROR:
imposible truncar archivo [n_inodo: %d  longitud: %d].\n",in1.n_inodo,in1.longitud-
sizeof(entrada));
                              return -1;
                        }
                        if (liberar_inodo(n_inodo) == -1) {
                              printf("(directorio.c|mi_unlink_directorio) ERROR:
imposible liberar inodo %d.\n",n_inodo);
                              return -1;
                        }
                        return 0;
                  }
                  i++;
            }
            posicion = posicion + sizeof(entrada);
      }
      return -1;
}

int mi_ls_directorio (char *nombre, char *buffer)
{
      entrada e;
      int n_inodo = 0;
      int posicion = 0;
      char nombre_tmp[MAX_NOMBRE];
      char buffer_tmp[sizeof(entrada)];
      inodo in;
      int leido = 0;
      if (leer_inodo(n_inodo,&in) == -1) {
            printf("(directorio.c|mi_ls_directorio) ERROR: imposible leer inodo %
d.\n",n_inodo);
            return -1;
      }
      // buscamos el inodo del directorio
      if (buscar_directorio(nombre,in,&n_inodo,0,nombre_tmp) == -1) {
            printf("(directorio.c|mi_ls_directorio) ERROR: imposible encontrar
directorio.\n");
            return -1;
      }
      // existe el directorio?
      if ((in.n_inodo == n_inodo) && (strlen(nombre_tmp)>1)) {
            printf("(directorio.c|mi_ls_directorio) ERROR: el directorio no
existe.\n");
            return -1;
      }
      if (leer_inodo(n_inodo,&in) == -1) {
            printf("(directorio.c|mi_ls_directorio) ERROR: imposible leer inodo %
d.\n",n_inodo);
            return -1;
      }
```

```
	// leemos las entradas del directorio e imprimimos la información
	while (posicion < in.longitud) {
		if ((leido = mi_read_ficheros(in.n_inodo,posicion,sizeof(entrada),
buffer_tmp)) < sizeof(entrada)) {
			printf("(directorio.c|mi_ls_directorio) ERROR: lectura incompleta
[n_inodo: %d posicion: %d  tamaño: %d  leido: %d].\n",in.n_inodo,posicion,sizeof
(entrada));
			return -1;
		}
		memcpy(&e,buffer_tmp,sizeof(entrada));
		strcat(buffer,e.nombre);
		strcat(buffer,":");
		posicion = posicion + sizeof(entrada);
	}
	// devolvemos el número de ficheros del directorio
	return in.longitud/sizeof(entrada);
}

int mi_mkdir_directorio (char *nombre)
{
	entrada e;
	int n_inodo = 0;
	inodo in1,in2;
	char nombre_tmp[MAX_NOMBRE];
	char buffer[sizeof(entrada)];
	superbloque sb;
	int escrito = 0;
	if (leer_inodo(n_inodo,&in1) == -1) {
		printf("(directorio.c|mi_mkdir_directorio) ERROR: imposible leer inodo
%d.\n",n_inodo);
		return -1;
	}
	// buscamos el inodo para comprobar que no existe
	if (buscar_directorio(nombre,in1,&n_inodo,0,nombre_tmp) == -1) {
		printf("(directorio.c|mi_mkdir_directorio) ERROR: imposible encontrar
directorio.\n");
		return -1;
	}
	// existe?
	if (in1.n_inodo != n_inodo) {
		printf("(directorio.c|mi_mkdir_directorio) ERROR: el directorio ya
existe.\n");
		return -1;
	}
	strcpy(e.nombre,nombre_tmp);
	e.n_inodo = reservar_inodo();
	if (leer_inodo(e.n_inodo,&in2) == -1) {
		printf("(directorio.c|mi_mkdir_directorio) ERROR: imposible leer inodo
%d.\n",e.n_inodo);
		return -1;
	}
	in2.tipo = 2;
	if (escribir_inodo(e.n_inodo,&in2) == -1) {
		printf("(directorio.c|mi_mkdir_directorio) ERROR: imposible escribir
inodo %d.\n",e.n_inodo);
		return -1;
	}
	memcpy(buffer,&e,sizeof(entrada));
	if ((escrito = mi_write_ficheros(in1.n_inodo,in1.longitud,sizeof(entrada),
buffer)) < sizeof(entrada)) {
```

```
            printf("(directorio.c|mi_mkdir_directorio) ERROR: escritura incompleta
[n_inodo: %d posicion: %d  tamaño: %d  escrito: %d].
\n",in1.n_inodo,in1.longitud,sizeof(entrada),escrito);
      }
      return 0;
}

int mi_write_directorio (char *nombre, int posicion, int tamano, char *buffer)
{
      inodo in;
      int n_inodo = 0;
      char nombre_tmp[MAX_NOMBRE];
      int escrito = 0;
      if (leer_inodo(0,&in) == -1) {
            printf("(directorio.c|mi_write_directorio) ERROR: imposible leer inodo
0.\n");
            return -1;
      }
      // buscamos el inodo del fichero
      if (buscar_directorio(nombre,in,&n_inodo,0,nombre_tmp) == -1) {
            printf("(directorio.c|mi_write_directorio) ERROR: imposible encontrar
directorio.\n");
            return -1;
      }
      if ((escrito = mi_write_ficheros(n_inodo,posicion,tamano,buffer)) < 0) {
            printf("(directorio.c|mi_write_directorio) ERROR: escritura incompleta
[n_inodo: %d posicion: %d  tamaño: %d  escrito: %d].
\n",n_inodo,posicion,tamano,escrito);
            return escrito;
      }
      return escrito;
}

int mi_stat_directorio (char *nombre, char *buffer)
{
      estat st;
      inodo in;
      int n_inodo = 0;
      char nombre_tmp[MAX_NOMBRE];
      if (leer_inodo(n_inodo,&in) == -1) {
            printf("(directorio.c|mi_stat_directorio) ERROR: imposible leer inodo
0.\n");
            return -1;
      }
      // buscamos el inodo del fichero
      if (buscar_directorio(nombre,in,&n_inodo,0,nombre_tmp) == -1) {
            printf("(directorio.c|mi_stat_directorio) ERROR: imposible encontrar
directorio.\n");
            return -1;
      }
      // existe?
      if ((in.n_inodo == n_inodo) && (n_inodo>0)) {
            printf("(directorio.c|mi_stat_directorio) ERROR: el archivo no
existe.\n");
            return -1;
      }
      mi_stat_ficheros(n_inodo,&st);
      memcpy(buffer,&st,sizeof(estat));
      return 0;
}
```

```
int mi_cat_directorio (char *nombre, int parte, char *buffer)
{
      inodo in;
      int n_inodo = 0;
      char nombre_tmp[MAX_NOMBRE];
      if (leer_inodo(n_inodo,&in) == -1) {
            printf("(directorio.c|mi_cat_directorio) ERROR: imposible leer inodo
0.\n");
            return -1;
      }
      // buscamos el inodo del fichero
      if (buscar_directorio(nombre,in,&n_inodo,0,nombre_tmp) == -1) {
            printf("(directorio.c|mi_cat_directorio) ERROR: imposible encontrar
directorio.\n");
            return -1;
      }
      // existe?
      if (in.n_inodo == n_inodo) {
            printf("(directorio.c|mi_cat_directorio) ERROR: el archivo no
existe.\n");
            return -1;
      }
      if (leer_inodo(n_inodo,&in) == -1) {
            printf("(directorio.c|mi_cat_directorio) ERROR: imposible leer inodo %
d.\n",n_inodo);
            return -1;
      }
      int tmp;
      if ((tmp = mi_read_ficheros(n_inodo,parte*TAM_BLOQUE,TAM_BLOQUE,buffer)) <
TAM_BLOQUE) {
            // marcamos el final
            buffer[tmp] = '#';
      }
}
```

directorio.h

```
#include "def.h"

// definición de la estructura de entradas de directorio
typedef struct {
      char nombre[MAX_NOMBRE];
      int n_inodo;
} entrada;

int montar (char *);
int desmontar ();
int mi_create_directorio (char *);
int mi_read_directorio (char *, int, int, char *);
int mi_write_directorio (char *, int, int, char *);
int mi_unlink_directorio (char *);
int mi_ls_directorio (char *, char *);
int mi_mkdir_directorio (char *);
int mi_stat_directorio (char *, char *);
int mi_cat_directorio (char *, int, char *);
```

ficheros_basico.c

```
#include <stdlib.h>
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
#include <sys/types.h>
#include "def.h"
#include "bloques.h"
#include "ficheros_basico.h"
#include "semaforos.h"

static superbloque *sb;
static char *mb;
int shmd_mb,shmd_sb;
int mutex;

// leemos el superbloque desde disco y lo escribimos en la memoria compartida
int leer_sb_sf (void)
{
      char buffer[TAM_BLOQUE];
      esperar_sem(mutex,0,0);
      if (leer_bloque(0,buffer) == -1) {
            printf("(ficheros_basico.c|leer_sb_sf) ERROR: error al leer el
bloque.\n");
            return -1;
      }
      memcpy(sb,buffer,sizeof(superbloque));
      senalizar_sem(mutex,0);
      if (DEBUG == 1) {
            printf("(ficheros_basico.c|leer_sb_sf) DEBUG: superbloque leído del
sistema de ficheros.\n");
      }
      return 0;
}

// escribimos el superbloque desde la memoria compartida a disco
int escribir_sb_sf ()
{
      char buffer[TAM_BLOQUE];
      memset(buffer,0,TAM_BLOQUE);
      esperar_sem(mutex,0,0);
      memcpy(buffer,sb,sizeof(superbloque));
      if (escribir_bloque(0,buffer) == -1) {
            printf("(ficheros_basico.c|escribir_sb_sf) ERROR: error al escribir el
bloque.\n");
            senalizar_sem(mutex,0);
            return -1;
      } else {
            senalizar_sem(mutex,0);
            if (DEBUG == 1) {
                  printf("(ficheros_basico.c|escribir_sb_sf) DEBUG: superbloque
leído del sistema de ficheros.\n");
            }
            return 0;
      }
}

// copia el superbloque de la memoria compartida a una variable dada
int leer_sb (superbloque *sb_a)
{
      memcpy(sb_a,sb,sizeof(superbloque));
      if (DEBUG == 1) {
            printf("(ficheros_basico.c|leer_sb) DEBUG: superbloque leído de
memoria.\n");
```

```
	}
	return 0;
}

// copia el superbloque dado a la memoria compartida
int escribir_sb (superbloque *sb_a)
{
	memcpy(sb,sb_a,sizeof(superbloque));
	if (DEBUG == 1) {
		printf("(ficheros_basico.c|escribir_sb) DEBUG: superbloque escrito en
memoria y disco.\n");
	}
	return 0;
}

// crea e inicializa la memoria compartida para el superbloque y el mapa de bits
int crear_shm (void)
{
	// memoria compartida para el superbloque
	shmd_sb = shmget(IPC_PRIVATE,sizeof(superbloque),0666);
	if (shmd_sb < 0) {
		printf("(ficheros_basico.c|crear_shm) ERROR: Imposible crear la memoria
compartida.\n");
		return -1;
	}
	sb = (superbloque *) shmat(shmd_sb,NULL,0);
	if (DEBUG == 1) {
		printf("(ficheros_basico.c|crear_shm) DEBUG: memoria compartida para el
superbloque creada [id: %d].\n",shmd_sb);
	}
	// inicializo la variable compartida sb
	leer_sb_sf();
	// memoria compartida para el mapa de bits
	shmd_mb = shmget(IPC_PRIVATE,div_c(sb->n_bloques,8*TAM_BLOQUE),0666);
	if (shmd_mb < 0) {
		printf("(ficheros_basico.c|crear_shm) ERROR: Imposible crear la memoria
compartida.\n");
		return -1;
	}
	mb = (char *) shmat(shmd_mb,NULL,0);
	if (DEBUG == 1) {
		printf("(ficheros_basico.c|crear_shm) DEBUG: memoria compartida para el
mapa de bits creada [id: %d].\n",shmd_mb);
	}
	// inicializo el mapa de bits compartido
	int i;
	char buffer[TAM_BLOQUE];
	for (i=0;i<sb->n_bloques_mb;i++) {
		memset(buffer,0,TAM_BLOQUE);
		leer_bloque(i+sb->n_bloque_mb,buffer);
		memcpy(mb+(i*TAM_BLOQUE),buffer,TAM_BLOQUE);
	}
	return 0;
}

// elimina la memoria compartida para el superbloque y para el mapa de bits
void remover_shm ()
{
	shmctl(shmd_sb,IPC_RMID,0);
	shmctl(shmd_mb,IPC_RMID,0);
```

```
	if (DEBUG == 1) {
		printf("(ficheros_basico.c|remover_shm) DEBUG: memoria compartida
eliminada.\n");
	}
}

// monta el sistema de ficheros, crea la memoria compartida e inicializa el mutex
int montar_basico (char *nombre)
{
	if (montar_bloques(nombre) == -1) {
		printf("(ficheros_basico.c|montar_basico) ERROR: problema al montar el
sistema de ficheros [nombre: %s].\n",nombre);
		return -1;
	}
	if (DEBUG == 1) {
		printf("(ficheros_basico.c|montar_basico) DEBUG: sistema de ficheros
montado.\n");
	}
	if (crear_shm() == -1) {
		printf("(ficheros_basico.c|montar_basico) ERROR: problema al crear la
memoria compartida.\n");
		return -1;
	}
	mutex = inicializar_mutex(1);
	if (DEBUG == 1) {
		printf("(ficheros_basico.c|montar_basico) DEBUG: memoria compartida
creada.\n");
	}
	return 0;
}

/* vuelca en disco la memoria compartida, desmonta el sistema de ficheros y
 * elimina la memoria compartida y el semáforo
 */
int desmontar_basico (void)
{
	escribir_sb_sf();
	escribir_mb_sf();
	if (desmontar_bloques() == -1) {
		printf("(ficheros_basico.c|desmontar_basico) ERROR: problema al
desmontar el sistema de ficheros.\n");
		return -1;
	}
	remover_shm();
	eliminar_sem(mutex);
	if (DEBUG == 1) {
		printf("(ficheros_basico.c|desmontar_basico) DEBUG: superbloque y mapa
de bits escritos en disco, sistema de ficheros desmontado y memoria compartida
eliminada correctamente.\n");
	}
	return 0;
}

// lee el inodo n_inodo y lo guarda en la variable apuntada por in
int leer_inodo (int n_inodo, inodo *in)
{
	int n_bloque = sb->n_bloque_inodos + (n_inodo/(TAM_BLOQUE/sizeof(inodo)));
	int n_inodo_bloque = n_inodo % (TAM_BLOQUE/sizeof(inodo));
	char buffer[TAM_BLOQUE];
	memset(buffer,0,TAM_BLOQUE);
```

```
	esperar_sem(mutex,0,0);
	if (leer_bloque(n_bloque,buffer) == -1) {
		printf("(ficheros_basico.c|leer_inodo) ERROR: imposible leer el inodo %
d en el bloque %d.\n",n_inodo,n_bloque);
		return -1;
	}
	memcpy(in,&buffer[n_inodo_bloque*sizeof(inodo)],sizeof(inodo));
	senalizar_sem(mutex,0);
	if (DEBUG == 1) {
		printf("((ficheros_basico.c|leer_inodo) DEBUG: leido inodo %d [n_inodo:
%d  siguiente: %d  tipo: %c  longitud: %d].\n",n_inodo,in->n_inodo,in->punteros[0],
in->tipo,in->longitud);
	}
	return 0;
}

// escribe el inodo apuntado por in en el inodo n_inodo
int escribir_inodo (int n_inodo, inodo *in)
{
	int n_bloque = sb->n_bloque_inodos + (n_inodo/(TAM_BLOQUE/sizeof(inodo)));
	int n_inodo_bloque = n_inodo % (TAM_BLOQUE/sizeof(inodo));
	char buffer[TAM_BLOQUE];
	memset(buffer,0,TAM_BLOQUE);
	esperar_sem(mutex,0,0);
	if (leer_bloque(n_bloque,buffer) == -1) {
		printf("(ficheros_basico.c|escribir_inodo) ERROR: imposible leer el
bloque %d.\n",n_bloque);
		return -1;
	}
	memcpy(&buffer[n_inodo_bloque*sizeof(inodo)],in,sizeof(inodo));
	if (escribir_bloque(n_bloque,buffer) == -1) {
		printf("(ficheros_basico.c|escribir_inodo) ERROR: imposible escribir el
inodo %d en el bloque %d.\n",n_inodo,n_bloque);
		return -1;
	}
	senalizar_sem(mutex,0);
	if (DEBUG == 1) {
		printf("(ficheros_basico.c|escribir_inodo) DEBUG: escrito inodo %d en
el bloque %d.\n",n_inodo,n_bloque);
	}
	return 0;
}

// libera el inodo de un fichero y se pone en la cola de inodos libres
int liberar_inodo (int n_inodo)
{
	inodo in;
	int i;
	if (leer_inodo(n_inodo,&in) == -1) {
		printf("(ficheros_basico.c|liberar_inodo) ERROR: imposible leer inodo %
d.\n",n_inodo);
		return -1;
	}
	// liberamos los bloques
	for (i=0;i<MAX_PTROS;i++) {
		if (in.punteros[i] > 0) {
			if (escribir_mb(in.punteros[i],0) == -1) {
				printf("(ficheros_basico.c|liberar_inodo) ERROR: imposible
liberar en el mapa de bits el bloque %d.\n",in.punteros[i]);
				return -1;
```

```
                  }
                  in.punteros[i] = -1;
            }
      }
      // liberamos el inodo
      in.tipo = '0';
      in.f_modificacion = time(NULL);
      in.f_ultimoacceso = time(NULL);
      in.longitud = 0;
      esperar_sem(mutex,0,0);
      // lo añadimos a la cola
      in.punteros[0] = sb->l_inodos;
      sb->l_inodos = n_inodo;
      if (DEBUG == 1) {
            printf("(ficheros_basico.c|liberar_inodo) DEBUG: inodo %d liberado
[n_inodo: %d  siguiente: %d].\n",n_inodo,in.n_inodo,in.punteros[0]);
      }
      if (escribir_inodo(n_inodo,&in) == -1) {
            printf("(ficheros_basico.c|liberar_inodo) ERROR: imposible escribir el
inodo %d.\n",n_inodo);
            return -1;
      }
      return 0;
}

// leo el valor del mapa de bits (memoria compartida) para el bloque n_bloque
int leer_mb (int n_bloque)
{
      int n_bit = n_bloque % 8;
      esperar_sem(mutex,0,0);
      if ((*(mb+(n_bloque/8)) & (128>>n_bit)) != 0) {
            senalizar_sem(mutex,0);
            if (DEBUG == 1) {
                  printf("(ficheros_basico.c|leer_mb) DEBUG: bloque %d
ocupado.\n",n_bloque);
            }
            // bloque ocupado
            return 1;
      } else {
            senalizar_sem(mutex,0);
            if (DEBUG == 1) {
                  printf("(ficheros_basico.c|leer_mb) DEBUG: bloque %d
libre.\n",n_bloque);
            }
            // bloque libre
            return 0;
      }
}

// escribo en el mapa de bits (memoria compartida) el valor para el bloque dado
int escribir_mb (int n_bloque, int valor)
{
      int n_bit = n_bloque % 8;
      if (valor == 0) {
            if (DEBUG == 1) {
                  printf("(ficheros_basico.c|escribir_mb) DEBUG: liberando
bloque.\n",n_bloque);
            }
            esperar_sem(mutex,0,0);
            *(mb+(n_bloque/8)) = *(mb+(n_bloque/8)) & (~(128>>n_bit));
```

```
            senalizar_sem(mutex,0);
      } else if (valor == 1) {
            if (DEBUG == 1) {
                  printf("(ficheros_basico.c|escribir_mb) DEBUG: ocupando
bloque.\n",n_bloque);
            }
            esperar_sem(mutex,0,0);
            *(mb+(n_bloque/8)) = *(mb+(n_bloque/8)) | (128>>n_bit);
            senalizar_sem(mutex,0);
      } else {
            printf("(ficheros_basico.c|escribir_mb) ERROR: valor %d
imposible.\n",valor);
            return -1;
      }
      return 0;
}

// vuelca el contenido del mapa de bits (memoria compartida) a disco
int escribir_mb_sf (void)
{
      int i;
      esperar_sem(mutex,0,0);
      for (i=0;i<sb->n_bloques_mb;i++) {
            if (escribir_bloque(i+sb->n_bloque_mb,&mb[i*TAM_BLOQUE]) == -1) {
                  printf("(ficheros_basico.c|escribir_mb_sf) ERROR: imposible
escribir bloque %d.\n",i+sb->n_bloque_mb);
                  senalizar_sem(mutex,0);
                  return -1;
            }
            if (DEBUG == 1) {
                  printf("(ficheros_basico.c|escribir_mb_sf) DEBUG: escrito bloque
%d correspondiente al mapa de bits.\n",i+sb->n_bloque_mb);
            }
      }
      senalizar_sem(mutex,0);
      return 0;
}

// cogemos un inodo de la lista de libres y lo marcamos como ocupado
int reservar_inodo (void)
{
      inodo in;
      int n_inodo;
      n_inodo = sb->l_inodos;
      if (n_inodo == -1) {
            printf("(ficheros_basico.c|reservar_inodo) ERROR: todos los inodos
ocupados.\n");
      }
      if (leer_inodo(n_inodo,&in) == -1) {
            printf("(ficheros_basico.c|reservar_inodo) ERROR: imposible leer inodo
%d.\n",n_inodo);
            return -1;
      }
      // modificamos la lista de inodos libres
      sb->l_inodos = in.punteros[0];
      in.punteros[0] = -1;
      in.f_ultimoacceso = in.f_modificacion = time(NULL);
      if (escribir_inodo(n_inodo,&in) == -1) {
            return -1;
      }
```

```
	if (DEBUG == 1) {
		printf("(ficheros_basico.c|reservar_inodo) DEBUG: inodo %d
reservado.\n",in.n_inodo);
	}
	return n_inodo;
}

// reservamos un bloque libre
int reservar_bloque (void)
{
	int n_bloque;
	// recorremos el mapa de bits
	for (n_bloque=sb->n_bloque_datos;n_bloque<sb->n_bloques;n_bloque++) {
		if (DEBUG == 1) {
			printf("(ficheros_basico.c|reservar_bloque) DEBUG: analizando
bloque %d.\n",n_bloque);
		}
		if (leer_mb(n_bloque) == 0) {
			if (escribir_mb(n_bloque,1) == -1) {
				printf("(ficheros_basico.c|reservar_bloque) ERROR:
imposible ocupar el bloque %d.\n",n_bloque);
			}
			if (DEBUG == 1) {
				printf("(ficheros_basico.c|reservar_bloque) DEBUG: bloque
reservado %d.\n",n_bloque);
			}
			return n_bloque;
		}
	}
	printf("(ficheros_basico.c|reservar_bloque) ERROR: sistema de ficheros
lleno.\n");
	// todos los bloques ocupados
	return -1;
}

// liberamos un bloque
int liberar_bloque (int n_bloque)
{
	if (escribir_mb(n_bloque,0) == -1) {
		printf("(ficheros_basico.c|liberar_bloque) ERROR: imposible liberar el
bloque %d.\n",n_bloque);
		return -1;
	} else {
		if (DEBUG == 1) {
			printf("(ficheros_basico.c|liberar_bloque) DEBUG: liberado bloque
%d.\n",n_bloque);
		}
		return 0;
	}
}

// división cociente implementada para determinadas operaciones
int div_c (int num, int den)
{
	if (num % den == 0) {
		return num/den;
	} else {
		return (num/den)+1;
	}
}
```

ficheros_basico.h

```
#include <time.h>
#include "def.h"

// definición de la estructura de superbloque
typedef struct {
      int tam_bloque;
      int n_bloques;
      int n_bloque_sb;
      int n_bloque_mb;
      int n_bloques_mb;
      int n_bloque_inodos;
      int n_bloques_inodos;
      int n_bloque_datos;
      int l_inodos;
} superbloque;

// definición de la estructura de inodo
typedef struct {
      char tipo; // 0: libre  1: fichero  2: directorio
      time_t f_creacion;
      time_t f_modificacion;
      time_t f_ultimoacceso;
      int longitud;
      int n_inodo;
      int punteros[MAX_PTROS];
} inodo;

int leer_sb_sf (void);
int escribir_sb_sf (void);
int leer_sb (superbloque *);
int escribir_sb (superbloque *);
int montar (char *);
int desmontar (void);
int leer_inodo (int, inodo *);
int escribir_inodo (int, inodo *);
int leer_mb (int);
int escribir_mb (int, int);
int escribir_mb_sf (void);
int reservar_inodo (void);
int reservar_bloque (void);
int liberar_bloque (int);
int div_c (int, int);
```

ficheros.c

```
#include <string.h>
#include "def.h"
#include "ficheros_basico.h"
#include "ficheros.h"

int mi_read_ficheros (int n_inodo, int posicion, int tamano, char *buffer) {
      inodo in;
      int leido = 0;
      int n_bloque_p = posicion/TAM_BLOQUE;
      int n_bloque_f = (posicion+tamano-1)/TAM_BLOQUE;
      int n_byte_p = posicion%TAM_BLOQUE;
      int n_byte_f = (posicion+tamano-1)%TAM_BLOQUE;
      char buffer_tmp[TAM_BLOQUE];
```

```
      memset(buffer_tmp,0,TAM_BLOQUE);
      if (DEBUG == 1) {
            printf("(ficheros.c|mi_read_ficheros) DEBUG: entrando en la
función.\n");
      }
      if (leer_inodo(n_inodo,&in) == -1) {
            printf("(ficheros.c|mi_read_ficheros) ERROR: imposible leer el inodo %
d.\n",n_inodo);
            return -1;
      }
      // si tenemos que leer un único bloque
      if (n_bloque_p == n_bloque_f) {
            if (leer_bloque(in.punteros[n_bloque_p],buffer_tmp) == -1) {
                  printf("(ficheros.c|mi_read_ficheros) ERROR: imposible leer el
bloque %d.\n",in.punteros[n_bloque_p]);
                  return -1;
            }
            if (DEBUG == 1) {
                  printf("(ficheros.c|mi_read_ficheros) DEBUG: leido bloque %
d.\n",in.punteros[n_bloque_p]);
            }
            memcpy(buffer,&buffer_tmp[n_byte_p],tamano);
            leido = tamano;
      } else {
            // leemos el primero
            if (leer_bloque(in.punteros[n_bloque_p],buffer_tmp) == -1) {
                  printf("(ficheros.c|mi_read_ficheros) ERROR: imposible leer el
bloque %d.\n",in.punteros[n_bloque_p]);
                  return -1;
            }
            if (DEBUG == 1) {
                  printf("(ficheros.c|mi_read_ficheros) DEBUG: leido bloque %
d.\n",in.punteros[n_bloque_p]);
            }
            memcpy(buffer,&buffer_tmp[n_byte_p],TAM_BLOQUE-n_byte_p);
            leido = TAM_BLOQUE-n_byte_p;
            // leemos los siguientes
            int n_bloque;
            for (n_bloque=n_bloque_p+1;n_bloque<n_bloque_f;n_bloque++) {
                  if (leer_bloque(in.punteros[n_bloque],buffer_tmp) == -1) {
                        printf("(ficheros.c|mi_read_ficheros) ERROR: imposible leer
el bloque %d.\n",in.punteros[n_bloque]);
                        return -1;
                  }
                  if (DEBUG == 1) {
                        printf("(ficheros.c|mi_read_ficheros) DEBUG: leido bloque %
d.\n",in.punteros[n_bloque]);
                  }
                  memcpy(buffer+leido,buffer_tmp,TAM_BLOQUE);
                  leido = leido+TAM_BLOQUE;
            }
            // leemos el ultimo
            if (leer_bloque(in.punteros[n_bloque_f],buffer_tmp) == -1) {
                  printf("(ficheros.c|mi_read_ficheros) ERROR: imposible leer el
bloque %d.\n",in.punteros[n_bloque_f]);
                  return -1;
            }
            memcpy(buffer+leido,buffer_tmp,n_byte_f);
            leido = leido+n_byte_f;
      }
```

```
	// modificamos los metadatos
	in.f_ultimoacceso = time(NULL);
	if (escribir_inodo(n_inodo,&in) == -1) {
		printf("(ficheros.c|mi_read_ficheros) ERROR: imposible escribir inodo %
d.\n",n_inodo);
		return -1;
	}
	if (DEBUG == 1) {
		printf("(ficheros.c|mi_read_ficheros) DEBUG: total leido %d.\n",leido);
		printf("(ficheros.c|mi_read_ficheros) DEBUG: saliendo de la
función.\n");
	}
	return leido;
}


int mi_write_ficheros (int n_inodo, int posicion, int tamano, char *buffer)
{
	inodo in;
	int escrito = 0;
	int n_bloque_p = posicion/TAM_BLOQUE;
	int n_bloque_f = (posicion+tamano-1)/TAM_BLOQUE;
	int n_byte_p = posicion%TAM_BLOQUE;
	int n_byte_f = (posicion+tamano-1)%TAM_BLOQUE;
	char buffer_tmp[TAM_BLOQUE];
	memset(buffer_tmp,0,TAM_BLOQUE);
	if (leer_inodo(n_inodo,&in) == -1) {
		printf("(ficheros.c|mi_write_ficheros) ERROR: imposible leer inodo %
d.\n",n_inodo);
		return -1;
	}
	// si tenemos que escribir un único bloque
	if (n_bloque_p == n_bloque_f) {
		if (in.punteros[n_bloque_p] == -1) {
			in.punteros[n_bloque_p] = reservar_bloque();
		}
		if (leer_bloque(in.punteros[n_bloque_p],buffer_tmp) == -1) {
			printf("(ficheros.c|mi_write_ficheros) ERROR: imposible leer el
bloque %d.\n",in.punteros[n_bloque_p]);
			return -1;
		}
		memcpy(buffer_tmp+n_byte_p,buffer,tamano);
		if (escribir_bloque(in.punteros[n_bloque_p],buffer_tmp) == -1) {
			printf("(ficheros.c|mi_write_ficheros) ERROR: imposible escribir
el bloque %d.\n",in.punteros[n_bloque_p]);
			return -1;
		}
		if (DEBUG == 1) {
			printf("(ficheros.c|mi_write_ficheros) DEBUG: escrito bloque %
d.\n",in.punteros[n_bloque_p]);
		}
		escrito = tamano;
	} else {
		// escribimos el primero
		// comprobamos si necesitamos un bloque libre
		if (in.punteros[n_bloque_p] == -1) {
			in.punteros[n_bloque_p] = reservar_bloque();
		}
		if (leer_bloque(in.punteros[n_bloque_p],buffer_tmp) == -1) {
			printf("(ficheros.c|mi_write_ficheros) ERROR: imposible leer el
```

```
bloque %d.\n",in.punteros[n_bloque_p]);
                return -1;
            }
            memcpy(buffer_tmp+n_byte_p,buffer,TAM_BLOQUE-n_byte_p);
            if (escribir_bloque(in.punteros[n_bloque_p],buffer_tmp) == -1) {
                printf("(ficheros.c|mi_write_ficheros) ERROR: imposible escribir
el bloque %d.\n",in.punteros[n_bloque_p]);
                return -1;
            }
            if (DEBUG == 1) {
                printf("(ficheros.c|mi_write_ficheros) DEBUG: escrito bloque %
d.\n",in.punteros[n_bloque_p]);
            }
            escrito = TAM_BLOQUE-n_byte_p;
            // escribimos los siguientes
            int n_bloque;
            for (n_bloque=n_bloque_p+1;n_bloque<n_bloque_f;n_bloque++) {
                memcpy(buffer_tmp,buffer+escrito,TAM_BLOQUE);
                // comprobamos si necesitamos un bloque libre
                if (in.punteros[n_bloque] == -1) {
                    in.punteros[n_bloque] = reservar_bloque();
                }
                if (escribir_bloque(in.punteros[n_bloque],buffer_tmp) == -1) {
                    printf("(ficheros.c|mi_write_ficheros) ERROR: imposible
escribir el bloque %d.\n",in.punteros[n_bloque]);
                    return -1;
                }
                if (DEBUG == 1) {
                    printf("(ficheros.c|mi_write_ficheros) DEBUG: escrito
bloque %d.\n",in.punteros[n_bloque]);
                }
                escrito = escrito+TAM_BLOQUE;
            }
            // escribimos el ultimo
            // comprobamos si necesitamos un bloque libre
            if (in.punteros[n_bloque_f] == -1) {
                in.punteros[n_bloque_f] = reservar_bloque();
            }
            if (leer_bloque(in.punteros[n_bloque_f],buffer_tmp) == -1) {
                printf("(ficheros.c|mi_write_ficheros) ERROR: imposible leer el
bloque %d.\n",in.punteros[n_bloque_f]);
                return -1;
            }
            memcpy(buffer_tmp,buffer+escrito,n_byte_f);
            if (escribir_bloque(in.punteros[n_bloque_f],buffer_tmp) == -1) {
                printf("(ficheros.c|mi_write_ficheros) ERROR: imposible escribir
el bloque %d.\n",in.punteros[n_bloque_f]);
                return -1;
            }
            if (DEBUG == 1) {
                printf("(ficheros.c|mi_write_ficheros) DEBUG: escrito bloque %
d.\n",in.punteros[n_bloque_f]);
            }
            escrito = escrito+n_byte_f;
      }
      if (in.longitud < posicion+tamano) {
            in.longitud = posicion+tamano;
      }
      // modificamos los meta-datos del fichero
      in.f_modificacion = in.f_ultimoacceso = time(NULL);
```

```
      if (escribir_inodo(n_inodo,&in) == -1) {
            printf("(ficheros.c|mi_write_ficheros) ERROR: imposible escribir inodo
%d.\n",n_inodo);
            return -1;
      }
      if (DEBUG == 1) {
            printf("(ficheros.c|mi_write_ficheros) DEBUG: total escrito %
d.\n",escrito);
      }
      return escrito;
}

int mi_truncar_ficheros (int n_inodo, int tamano)
{
      inodo in;
      if (tamano == 0) {
            if (liberar_inodo(n_inodo) == -1) {
                  return -1;
            }
      } else {
            if (leer_inodo(n_inodo,&in) == -1) {
                  return -1;
            }
            in.longitud = tamano;
            in.f_modificacion = time(NULL);
            in.f_ultimoacceso = time(NULL);
            int i;
            // liberamos los bloques innecesarios
            for (i=(tamano/TAM_BLOQUE)+1;i<(in.longitud/TAM_BLOQUE)+1;i++) {
                  if (escribir_mb(in.punteros[i],0) == -1) {
                        return -1;
                  }
                  in.punteros[i] = -1;
            }
            if (escribir_inodo(n_inodo,&in) == -1) {
                  return -1;
            }
      }
      return 0;
}

int mi_stat_ficheros (int n_inodo, estat *estado)
{
      inodo in;
      if (leer_inodo(n_inodo,&in) == -1) {
            return -1;
      }
      estado->n_inodo = n_inodo;
      estado->f_creacion = in.f_creacion;
      estado->f_modificacion = in.f_modificacion;
      estado->f_ultimoacceso = in.f_ultimoacceso;
      estado->longitud = in.longitud;
      return 0;
}
```

ficheros.h

```
#include <time.h>

// definición de la estructura de estadística
```

```
typedef struct {
      int n_inodo;
      time_t f_creacion;
      time_t f_modificacion;
      time_t f_ultimoacceso;
      int longitud;
} estat;

int mi_write_ficheros (int, int, int, char *);
int mi_read_ficheros (int, int, int, char*);
int mi_truncar_ficheros (int, int);
int mi_stat_ficheros (int, estat *);
```

mensajes.c

```
#include <sys/ipc.h>

// creamos una cola
int crear_cola (int id)
{
      int msg;
      if ((msg = msgget(id,0600|IPC_CREAT)) < 0) {
            printf("(mensajes.c) ERROR: imposible crear la cola.\n");
            return -1;
      }
      return msg;
}

// eliminamos una cola dado su id
void eliminar_cola (int id) {
      int msg;
      if ((msg = msgctl(id,IPC_RMID,0)) < 0) {
            printf("(mensajes.c) ERROR: imposible eliminar la cola [id: %d].
\n",id);
      }
}
```

mensajes.h

```
// definición del contenido del mensaje
typedef struct {
      int tipo;
      int tamano;
      int posicion;
      char nombre[MAX_NOMBRE];
      char info[TAM_BLOQUE];
} c_mensaje;

// definición de la estructura del mensaje
typedef struct {
      long pid;
      c_mensaje contenido;
} mensaje;

int crear_cola (int);
void eliminar_cola (int);
```

mi_mkfs.c

```
#include <string.h>
#include "def.h"
```

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

int main (int argc, char **argv)
{
      // comprobamos argumentos
      if (argc != 3) {
            printf ("(mi_mkfs.c) ERROR: usar %s nombre_sistema_fichero
numero_bloques.\n",argv[0]);
            return -1;
      }
      if (montar_bloques(argv[1]) == -1) {
            return -1;
      }
      int i;
      char buffer[TAM_BLOQUE];
      memset(buffer,0,TAM_BLOQUE);
      // inicializamos todo a cero
      for (i=0;i<atoi(argv[2]);i++) {
            if (escribir_bloque(i,buffer) == -1) {
                  return -1;
            }
      }
      // inicializamos el superbloque
      superbloque sb;
      sb.tam_bloque = TAM_BLOQUE;
      sb.n_bloques = atoi(argv[2]);
      sb.n_bloque_sb = 0;
      sb.n_bloque_mb = 1;
      sb.n_bloques_mb = div_c(sb.n_bloques,TAM_BLOQUE*8);
      sb.n_bloque_inodos = sb.n_bloques_mb + sb.n_bloque_mb;
      sb.n_bloques_inodos = div_c(sb.n_bloques,div_c(TAM_BLOQUE,sizeof(inodo)));
      sb.n_bloque_datos = sb.n_bloque_inodos + sb.n_bloques_inodos;
      sb.l_inodos = 1;
      memset(buffer,0,TAM_BLOQUE);
      memcpy(buffer,&sb,sizeof(superbloque));
      if (escribir_bloque(0,buffer) == -1) {
            return -1;
      }
      printf("(mi_mkfs.c) tamaño de bloques: %d\n",TAM_BLOQUE);
      printf("(mi_mkfs.c) número de bloques: %d\n",sb.n_bloques);
      printf("(mi_mkfs.c) bloque del SB: %d\n",sb.n_bloque_sb);
      printf("(mi_mkfs.c) bloque inicial del MB: %d\n",sb.n_bloque_mb);
      printf("(mi_mkfs.c) bloques para el MB: %d\n",sb.n_bloques_mb);
      printf("(mi_mkfs.c) bloque inicial de inodos: %d\n",sb.n_bloque_inodos);
      printf("(mi_mkfs.c) bloques para inodos: %d\n",sb.n_bloques_inodos);
      printf("(mi_mkfs.c) bloque inicial de datos: %d\n",sb.n_bloque_datos);
      printf("(mi_mkfs.c) primer inodo libre: %d\n",sb.l_inodos);
      // inicializamos inodos
      inodo in;
      in.tipo = '0';
      in.longitud = 0;
      for (i=0;i<MAX_PTROS;i++) {
            in.punteros[i] = -1;
      }
      int j;
      int n_inodo_tmp = 0;
      for (i=sb.n_bloque_inodos;i<sb.n_bloque_datos;i++) {
            memset(buffer,0,TAM_BLOQUE);
            for (j=0;j<(TAM_BLOQUE/sizeof(inodo));j++) {
```

```
                  in.f_creacion = in.f_modificacion = in.f_ultimoacceso = time
(NULL);
                  in.n_inodo = n_inodo_tmp;
                  // apuntamos al siguiente (cola de inodos libres)
                  if ((in.n_inodo != sb.n_bloques-1) && (in.n_inodo != 0)) {
                        in.punteros[0] = in.n_inodo+1;
                  // el inodo 0 es especial (contiene /)
                  } else if (in.n_inodo == 0) {
                        in.punteros[0] = -1;
                        in.tipo = '2';
                  // último inodo
                  } else {
                        in.punteros[0] = -1;
                  }
                  memcpy(&buffer[j*sizeof(inodo)],&in,sizeof(inodo));
                  n_inodo_tmp++;
            }
            if (escribir_bloque(i,buffer) == -1) {
                  return -1;
            }
      }
      if (desmontar_bloques() == -1) {
            return -1;
      }
      return 0;
}
```

semaforos.c

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>

static int pid = -1;
static int rec = 0;

int crear_sem (int num)
{
      int s;
      s = semget(IPC_PRIVATE,num,IPC_CREAT|0600);
      if (s<0) {
            printf("(semaforos.c) ERROR: imposible crear semáforo.\n");
            exit(1);
      }
      return s;
}

void eliminar_sem (int s)
{
      semctl(s,0,IPC_RMID,0);
}

void esperar_sem (int s, int pos, int flag)
{
      struct sembuf sbuf;
      if (getpid() != pid) {
            pid = getpid();
            sbuf.sem_num = pos;
            sbuf.sem_op = -1;
            sbuf.sem_flg = flag;
```

```
            semop(s,&sbuf,1);
            rec = 0;
      } else {
            rec++;
      }
}

void senalizar_sem (int s, int pos)
{
      struct sembuf sbuf;
      if (rec == 0) {
            sbuf.sem_num = pos;
            sbuf.sem_op = 1;
            sbuf.sem_flg = 0;
            semop(s,&sbuf,1);
            pid = -1;
      } else {
            rec--;
      }
}

void esperar_cero (int s, int pos)
{
      struct sembuf sbuf;
      sbuf.sem_num = pos;
      sbuf.sem_op = 0;
      sbuf.sem_flg = 0;
      semop(s,&sbuf,1);
}

int inicializar_mutex (int valor)
{
      int s;
      s = crear_sem(1);
      semctl(s,0,SETVAL,valor);
      return s;
}
```

semaforos.h

```
int crear_sem (int);
void eliminar_sem (int);
void esperar_sem (int, int, int);
void senalizar_sem (int, int);
void esperar_cero (int, int);
int inicializar_mutex (int);
```

servidor.c

```
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>
#include "def.h"
#include "mensajes.h"
#include "directorio.h"

// servidores que se encargan de comprobar los mensajes y realizar las operaciones
void servidor (int cola_s, int cola_r)
{
      mensaje msg1,msg2;
      int control;
      while (1) {
```

```
            if (msgrcv(cola_s,&msg1,sizeof(c_mensaje),0,0) < 0) {
                  printf("(servidor.c) ERROR: imposible recibir mensaje [%d].
\n",getpid());
            }
            // mi_create
            if (msg1.contenido.tipo == 0) {
                  if (mi_create_directorio(msg1.contenido.nombre) == -1) {
                        printf ("(servidor.c) ERROR: imposible crear fichero.\n");
                  }
                  if (DEBUG == 1) {
                        printf("(servidor.c) DEBUG: hecho mi_create_directorio(%s).
\n",msg1.contenido.nombre);
                  }
            // mi_write
            } else if (msg1.contenido.tipo == 1) {
                  if ((msg2.contenido.tamano = mi_write_directorio
(msg1.contenido.nombre,msg1.contenido.posicion,msg1.contenido.tamano,msg1.contenido
.info)) == -1) {
                        printf ("(servidor.c) ERROR: imposible escribir.\n");
                  }
                  if (DEBUG == 1) {
                        printf("(servidor.c) DEBUG: hecho mi_write_directorio(%s,%
d,%d,%s).
\n",msg1.contenido.nombre,msg1.contenido.posicion,msg1.contenido.tamano,msg1.conten
ido.info);
                  }
            // mi_ls
            } else if (msg1.contenido.tipo == 2) {
                  msg2.contenido.tamano = mi_ls_directorio
(msg1.contenido.nombre,msg1.contenido.info);
                  strcpy(msg2.contenido.info,msg1.contenido.info);
                  if (DEBUG == 1) {
                        printf("(servidor.c) DEBUG: hecho mi_ls_directorio(%s,%s).
\n",msg1.contenido.nombre,msg1.contenido.info);
                  }
            // mi_unlink
            } else if (msg1.contenido.tipo == 3) {
                  mi_unlink_directorio(msg1.contenido.nombre);
                  if (DEBUG == 1) {
                        printf("(servidor.c) DEBUG: hecho mi_unlink_directorio(%s).
\n",msg1.contenido.nombre);
                  }
            // mi_stat
            } else if (msg1.contenido.tipo == 4) {
                  mi_stat_directorio(msg1.contenido.nombre,msg1.contenido.info);
                  strcpy(msg2.contenido.info,msg1.contenido.info);
                  if (DEBUG == 1) {
                        printf("(servidor.c) DEBUG: hecho mi_stat_directorio(%s,%
s).\n",msg1.contenido.nombre,msg1.contenido.info);
                  }
            // mi_read
            } else if (msg1.contenido.tipo == 5) {
                  if ((msg2.contenido.tamano = mi_read_directorio
(msg1.contenido.nombre,msg1.contenido.posicion,msg1.contenido.tamano,msg1.contenido
.info)) == -1) {
                        printf("(servidor.c) ERROR: imposible leer.\n");
                  }
                  if (DEBUG == 1) {
                        printf("(servidor.c) DEBUG: hecho mi_read_directorio(%s,%
d,%d,%s).
```

```
\n",msg1.contenido.nombre,msg1.contenido.posicion,msg1.contenido.tamano,msg1.conten
ido.info);
                  }
            } else {
                  printf("(servidor.c) ERROR: tipo de operación no reconocida.\n");
            }
            msg2.pid = msg1.pid;
            if (msgsnd(cola_r,&msg2,sizeof(c_mensaje),0) < 0) {
                  printf("(servidor.c) ERROR: imposible enviar mensaje [%d].
\n",getpid());
            }
      }
}
```

servidor.h

```
void servidor (int, int);
```

simulador.c

```
#include <unistd.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/resource.h>
#include "def.h"
#include "cliente.h"
#include "mensajes.h"

int n_acabados;

void enterrador (void)
{
      int n_enterrados = 0;
      while (wait3(NULL,WNOHANG,NULL) > 0) {
            n_enterrados++;
            n_acabados++;
      }
      if (n_enterrados > 0) {
            printf("(simulador.c) Acabados: %d.\n",n_acabados);
      }
}

int main (void)
{
      n_acabados = 0;
      int n_clientes = 0;
      int cola_s,cola_r;
      // colas
      cola_s = crear_cola(50);
      cola_r = crear_cola(51);
      int i,pid_cliente;
      // creamos los clientes
      for (i=0;i<N_CLIENTES;i++) {
            if ((pid_cliente = fork()) == 0) {
                  cliente(cola_s,cola_r);
                  exit(0);
            } else if (pid_cliente > 0) {
                  n_clientes++;
                  usleep(1000000);
            }
```

```
	}
	printf("(simulador.c) Clientes creados: %d.\n",n_clientes);
	while (n_acabados < n_clientes) {
		enterrador();
	}
}
```

mi_cat.c

```
#include "def.h"
#include "ficheros.h"
#include "directorio.h"

int main (int argc, char **argv)
{
	if (argc != 3) {
		printf("(mi_cat.c) ERROR: usar %s nombre_sistema_fichero
nombre_fichero.\n",argv[0]);
		return -1;
	}
	estat st;
	if (montar(argv[1]) == -1) {
		return -1;
	}
	char buffer[sizeof(estat)];
	mi_stat_directorio(argv[2],buffer);
	memcpy(&st,buffer,sizeof(estat));
	int i,j;
	char buffer_tmp[TAM_BLOQUE];
	for (i=0;i<=(st.longitud/TAM_BLOQUE);i++) {
		mi_cat_directorio(argv[2],i,buffer_tmp);
		j = 0;
		// imprimimos hasta tener el carácter final
		while ((j<TAM_BLOQUE) && (buffer_tmp[j] != '#')) {
			printf("%c",buffer_tmp[j]);
			j++;
		}
	}
	if (desmontar() == -1) {
		return -1;
	}
	return 0;
}
```

mi_ls.c

```
#include "def.h"
#include "ficheros.h"
#include "directorio.h"

int main (int argc, char **argv)
{
	if (argc != 3) {
		printf("(mi_ls.c) ERROR: usar %s nombre_sistema_fichero
nombre_directorio.\n",argv[0]);
		return -1;
	}
	if (montar(argv[1]) == -1) {
		return -1;
	}
	char buffer[sizeof(estat)];
```

```
	if (mi_stat_directorio(argv[2],buffer) == -1) {
		return -1;
	}
	estat st;
	memcpy(&st,buffer,sizeof(estat));
	char buffer_tmp[MAX_NOMBRE*(st.longitud/sizeof(entrada))];
	memset(buffer_tmp,'\0',MAX_NOMBRE*(st.longitud/sizeof(entrada)));
	int n_f = mi_ls_directorio(argv[2],buffer_tmp);
	printf("(mi_ls.c) Número de ficheros: %d.\n\n",n_f);
	int i = 0;
	// imprimimos el resultado
	while (buffer_tmp[i] != '\0') {
		if (buffer_tmp[i] == ':') {
			printf("\n");
		} else {
			printf("%c",buffer_tmp[i]);
		}
		i++;
	}
	printf("\n");
	if (desmontar() == -1) {
		return -1;
	}
	return 0;
}
```

mi_rm.c

```
#include "directorio.h"

int main (int argc, char **argv)
{
	if (argc != 3) {
		printf("(mi_rm.c) ERROR: usar %s nombre_sistema_ficheros
nombre_fichero.\n",argv[0]);
		return -1;
	}
	if (montar(argv[1]) == -1) {
		return -1;
	}
	mi_unlink_directorio(argv[2]);
	if (desmontar() == -1) {
		return -1;
	}
}
```

mi_stat.c

```
#include "ficheros.h"
#include "directorio.h"

int main (int argc, char **argv)
{
	if (argc != 3) {
		printf("(mi_stat.c) ERROR: usar %s nombre_sistema_ficheros
nombre_fichero.\n",argv[0]);
		return -1;
	}
	estat st;
	char buffer[sizeof(estat)];
	if (montar(argv[1]) == -1) {
```

```
            return -1;
      }
      mi_stat_directorio(argv[2],buffer);
      memcpy(&st,buffer,sizeof(estat));
      printf("Inodo: %d\nFecha de creación: %sFecha de modificación: %sFecha de
último acceso: %sTamaño en bytes: %d.\n",st.n_inodo,ctime(&st.f_creacion),ctime
(&st.f_modificacion),ctime(&st.f_ultimoacceso),st.longitud);
      if (desmontar() == -1) {
            return -1;
      }
      return 0;
}
```

Otros

Copia del documento en http://bulma.net/~jander/uib/aso/