EmoFS: sistema de fitxers amb inodes com a pràctica per Ampliació de Sistemes Operatius (4520)

Bartomeu Miró Mateu <bartomeumiro@gmail.com>,
DNI 43182841-L, Informàtica Tècnica de Sistemes (TIS2)
Pau Rul·lan Ferragut <paurullan@gmail.com>,
DNI 43142697-X, Informàtica Tècnica de Sistemes (TIS2)

15 d'agost de 2009

Resum

Un sistema de fitxers és el component del sistema operatiu que facilita als usuaris desar informació i ordenar-ho per fitxers i directoris. Aquest document és la memòria d'un sistema de fitxers basat en inodes amb la finalitat d'experimentar el que s'ha estudiat a l'assignatura d'ampliació de sistemes operatius (codi 4520).

1 Descripció del problema

El temari de l'assignatura d'ampliació de sistemes operatius és principalment l'estudi de la memòria virtual i els sistemes de fitxers. Un sistema de fitxers és la part del sistema operatiu encarregat de mantenir permanentment les dades de l'usuari i els programes del sistema. Com a pràctica de l'assignatura s'implementa un petit sistema de fitxers basat en inodes i amb capacitat de directoris en arbre. Amb el llenguatge de programació C haurem d'escriure no sols les eines típiques del UNIX com crear un directori o esborrar un fitxer sinó també un simulador que executi una càrrega de treball contra el nostre sistema.

Tal i per seguir les tradicions del UNIX vàrem decidir anomenar el nostre sistema de fitxers (*SF* a partir d'ara) algunacosa-fs. Com que un dels membres del grup estava especialment trist en l'època de començar la pràctica usarem *emo* dels *emotive hardcore*, gent que segons la cultura

popular sempre està deprimida i en conseqüència anomenam al nostre SF *emofs*.

L'estructura general de la pràctica ve indicada per l'enunciat exposat pels professors: nou setmanes de feina on seqüencialment es van introduint distintes capes de la feina. Des de la primera setmana on cream una imatge i escrivim blocs d'informació fins a la darrera on treballam amb fitxers dins la nostra imatge de sistema de fitxers.

2 Disseny i estructura del SF

La pràctica en sí està dividida en quatre seccions: les biblioteques, les eines d'interacció, els jocs de proves i el simulador. Anomenam biblioteca a tot el codi que l'usuari final no usa directament. Aquesta secció és el major cos de la pràctica en nombre de línies perquè és la implementació del sistema de fitxers pròpiament dit.

Les altres tres seccions formen el suport per l'ús del sistema de fitxers. Anomenam a les eines d'interacció a tots els petits programes clàssics del UNIX que s'usen al dia a dia: `cd`, `cat`, `ls` o `mkdir`. El simulador és la peça indicada a l'enunciat que fa la prova del sistema. El simulador està completament definit i no és més que crear un gran nombre de clients que facin un directori i duguin a terme escriptures a un fitxer. Els jocs de proves són petits programes que tenen com a únic objectiu comprovar la correcció de cadascun dels mòduls.

L'estructura de les biblioteques és un conjunt de vàries capes més un fitxer comú a totes elles. L'estructura completa es pot estudiar a la taula 1 però bàsicament està formada pel nucli del sistema de fitxers, les llibreries d'accés al nucli, la gestió de fitxers i les eines d'usuari. Aquest element al que totes tenen accés s'anomena `common.h` i allà s'emmagatzemen variables globals de configuració com el grandària de bloc predeterminat. Les dades i valors que apareixen al fitxer comú poden modificar-se a la creació de la imatge del SF.

La informació permanent del sistema de fitxers està separada en dues parts: les *dades* i les seves *metadades*. Les metadades estan compostes pel *superbloc*, el *mapa de bits* i pel vector d'*inodes*. Aquest esquema es pot estudiar a la taula 2. El superbloc s'encarrega de mantenir informació com quants de blocs queden lliures o quin és el primer inode i el mapa de bits té com a funció indicar quins blocs de dades estan lliures i quins estan ocupats.

El vector d'inodes és un conjunt d'estructures que assenyalen les característiques d'un fitxer o directori: el seu tamany, la darrera modi-

Nivell	Components
simulació	sim.c
eines	mi_mkfs.c mi_rm.c mi_ln.c ...
fitxers	dir file inode
llibreries	block_lib
nucli	block super bitmap

Taula 1: *Esquema dels mòduls del SF*

Metadades			Dades
Superbloc	Mapa de bits	Inodes	Dades

Taula 2: *Esquema de l'ordenació de les dades del SF*

ficació, quants d'enllaços té i on es poden trobar els blocs físics on es desa la informació. Un inode es pot comportar com un fitxer de dades, on es desa informació, o com una entrada de directori, un text que sosté l'estructura d'arbre dels fitxers.

3 Implementació

Per simplicitat la gestió de concurrència es fa mitjançant un sol *mutex* que bloca totes les estructures. Aquest *biglock* implica un rendiment molt baix doncs sols permet una operació alhora però redueix enormement la complexitat i els possibles problemes d'accés simultani. Tots els controls es troben dins les funcions de `dir.h`. També creàrem unes eines de `mi_mount` i `mi_umount` que s'encarreguen de crear i destruir els semàfors. Aquestes eines s'han d'executar abans de començar la interacció amb el sistema de fitxers però no cal fer-ho pel simulador perquè el propi `sim.c` ja ho fa.

Les variables i característiques com els tamanys de bloc, el nombre d'entrades d'inode o el nombre de blocs de dades estan definits als fitxers corresponents: `common.h`, `super.h`, `inode.h` ... Podem trobar

que totes les dades indirectes, com el tamany de *padding* del superbloc o el nombre final d'entrades al mapa de bits són calculades mitjançant macros però recordant que s'ha recompilar el projecte si es vol fer algun canvi.

Com a funcions i tasques extres hem desenvolupat els programes d'usuari `mkdir`, `touch`, `append`. Amb aquests programes són molt útils per fer comprovacions de com està la imatge del sistema de fitxers. Alhora així podem escriure el simulador d'una manera especial: fent que cada tasca del fils usi les eines d'usuari normals, imitant més el comportament d'usuari.

El `ls` accepta varis camins com entrada. És l'únic programa que ho fa però sols ho hem volgut implementar com a prova conceptual: fer-ho per eines com `rm` sols era un poc més de feina. Aquest processament es podria haver fet amb una biblioteca d'arguments però l'hem fet a mà.

El programa `mi_ls` usa com a sortida un `emofs_extract_path` (el que a l'enunciat és un `info_fichero`) i aquestes funcions demanen un *buffer* per recórrer els fitxers d'un directori. Aquest *buffer* pot ser fixo, que malbarata l'espai, o dinàmic, amb el perill de quedar-se sense memòria. Nosaltres hem usat la memòria dinàmica per així recordar els usos de `malloc` i `realloc` sempre recordant que la pràctica no pretén tenir directoris amb un gran nombre d'entrades.

4 Jocs de proves i resultats

Els jocs de proves no són eines d'usuari perquè no tenen cap utilitat més enllà de fer les comprovacions de les feines setmana a setmana. Segons el que s'hagi treballat durant la creació d'un joc de proves potser s'usen directament les biblioteques del SF o alguna eina com el `mkdir`. El codi d'aquestes proves es pot veure a l'apèndix.

4.1 Execucions

Aquestes són les sortides de les execucions de les distintes eines d'usuari i el simulador. Les execucions estan fetes per ordre sobre la mateixa imatge de sistema de fitxers.

```
mi_mkfs
mi_mount
mi_touch /x
mi_mkdir /a
mi_mkdir /a/b
```

```
mi_touch /a/b/y

mi_ls /
Type   Size   Epoc   Name
d   64   1250506318   a
f   0   1250506278   x
d   0   1250506266   authors

mi_ls /a
Size   Epoc   Name
d   64   1250506372   b

mi_write /a/b/y "prova de text molt llarg"

mi_cat /a/b/y
prova de text molt llarg

mi_write /a/b/y "MES TEXT QUE SOBREPASI" 5

mi_cat /a/b/y
provaMES TEXT QUE SOBREPASI

mi_ln /a/b/y /z

mi_cat /z
provaMES TEXT QUE SOBREPASI

mi_rm /a/b/y

mi_ls /a/b
Type   Size   Epoc   Name

mi_umount
```

4.2 Simulador

```
Inici worker 13861
sim: worker: final client amb pid 13861
Inici worker 13862
sim: worker: final client amb pid 13862
Inici worker 13863
sim: worker: final client amb pid 13863
[...]
Inici worker 13959
sim: worker: final client amb pid 13959
Inici worker 13960
sim: worker: final client amb pid 13960
```

```
Type   Size   Epoc   Name
d   64   1250518506   process_13960
d   64   1250518506   process_13959
d   64   1250518506   process_13958
d   64   1250518506   process_13957
d   64   1250518506   process_13956
d   64   1250518506   process_13955
d   64   1250518506   process_13954
d   64   1250518506   process_13953
d   64   1250518506   process_13952
d   64   1250518506   process_13951
d   64   1250518506   process_13950
d   64   1250518506   process_13949
d   64   1250518506   process_13948
d   64   1250518506   process_13947
d   64   1250518506   process_13946
d   64   1250518506   process_13945
d   64   1250518506   process_13944
d   64   1250518506   process_13943
d   64   1250518506   process_13942
d   64   1250518506   process_13941
d   64   1250518506   process_13940
d   64   1250518506   process_13939
d   64   1250518506   process_13938
d   64   1250518506   process_13937
d   64   1250518506   process_13936
d   64   1250518506   process_13935
d   64   1250518506   process_13934
d   64   1250518506   process_13933
d   64   1250518506   process_13932
d   64   1250518506   process_13931
d   64   1250518506   process_13930
d   64   1250518506   process_13929
d   64   1250518506   process_13928
d   64   1250518506   process_13927
d   64   1250518506   process_13926
d   64   1250518506   process_13925
d   64   1250518506   process_13924
d   64   1250518506   process_13923
d   64   1250518506   process_13922
d   64   1250518506   process_13921
d   64   1250518506   process_13920
d   64   1250518506   process_13919
d   64   1250518506   process_13918
d   64   1250518506   process_13917
d   64   1250518506   process_13916
d   64   1250518506   process_13915
d   64   1250518506   process_13914
d   64   1250518506   process_13913
```

```
d   64   1250518506   process_13912
d   64   1250518506   process_13911
d   64   1250518506   process_13910
d   64   1250518506   process_13909
d   64   1250518506   process_13908
d   64   1250518506   process_13907
d   64   1250518506   process_13906
d   64   1250518506   process_13905
d   64   1250518506   process_13904
d   64   1250518506   process_13903
d   64   1250518506   process_13902
d   64   1250518506   process_13901
d   64   1250518506   process_13900
d   64   1250518506   process_13899
d   64   1250518506   process_13898
d   64   1250518506   process_13897
d   64   1250518506   process_13896
d   64   1250518506   process_13895
d   64   1250518506   process_13894
d   64   1250518506   process_13893
d   64   1250518506   process_13892
d   64   1250518506   process_13891
d   64   1250518506   process_13890
d   64   1250518506   process_13889
d   64   1250518506   process_13888
d   64   1250518506   process_13887
d   64   1250518506   process_13886
d   64   1250518506   process_13885
d   64   1250518505   process_13884
d   64   1250518505   process_13883
d   64   1250518505   process_13882
d   64   1250518505   process_13881
d   64   1250518505   process_13880
d   64   1250518505   process_13879
d   64   1250518505   process_13878
d   64   1250518505   process_13877
d   64   1250518505   process_13876
d   64   1250518505   process_13875
d   64   1250518505   process_13874
d   64   1250518505   process_13873
d   64   1250518505   process_13872
d   64   1250518505   process_13871
d   64   1250518505   process_13870
d   64   1250518505   process_13869
d   64   1250518505   process_13868
d   64   1250518505   process_13867
d   64   1250518505   process_13866
d   64   1250518505   process_13865
d   64   1250518505   process_13864
```

```
d   64   1250518505   process_13863
d   64   1250518501   process_13862
d   64   1250518501   process_13861

sim: anam a mostrar un fitxer dels creats
/simul_200981716151/process_13861/prueba.dat
--------------------------------------
16:15:1 escriptura nombre 35 a la posicio 14
mbre 14 a la posicio 38
16:15:1 escriptu16:15:1 escriptura nombre 11 a la posicio 113
16:15:1 escriptura nombre 1 a la posici16:15:1 escriptura nombre 4 a la posicio 275
16:1516:15:1 escriptura nombre 49 a la posicio 356
o 362
16:15:1 escript16:15:1 escriptura nombre 43 16:15:1 escriptura nombre 47 a la posic
16:15:1 escriptura nombre 42 a la posicio 556
6:15:1 escriptura nombre 27 a la posicio 601
osicio 612
la p16:15:1 escriptura nombre 15 a la posicio 662
16:15:1 escriptura nombre 12 a la posicio 755
16:15:1 escriptura nombre 48 a la posicio 809
16:15:1 escriptura n16:15:1 escriptura nombre 44 a la posicio 916:15:1 escriptura n
ra nombre 0 a la posicio 976
16:15:1 escriptura nombre 23 a la posicio 1034
16:15:1 escriptura nombre 46 a la posicio 1121
io 1129
 29 a la posicio 1151
sicio 1162
16:15:1 escriptura nombre 16 a la posicio 1232
16:15:1 escriptura nombre 38 a la posicio 1304
16:15:1 escriptura nombre 40 a la posicio 1351
riptura nombre 32 a la posicio 1387
16:15:1 escriptura nombre 19 a la posicio 1485
mbre 17 a la posicio 1511
 escrip16:15:1 escriptura nombre 21 a la posicio 1565
escriptura nombre 20 a la posicio 1604
16:15:1 escriptura nombre 34 a la posicio 1661
16:15:1 escriptura nombre 37 a la posicio 1758
o 1765
cio16:15:1 escriptura nombre 36 a la posicio 1815
cio 1824
16:15:1 escriptura nombre 33 a la posicio 1912
16:15:1 escriptura nombre 3 a la posicio 1970
--------------------------------------
Simulació acabada
```

5 Conclusions

Hem escrit des de zero un sistema de fitxers d'usuari amb inodes i directoris en arbre. Hem programat les eines mínimes per treballar sobre el nostre SF i executat una completa simulació per observar el seu comportament. Com a resultat tenim un sistema de fitxers que malgrat no l'usariem al nostre dia a dia té les característiques d'un sistema de fitxers clàssic. És una prova conceptual completa del funcionament d'una peça important dels nostres sistemes operatius.

D'aquesta manera tancam una pràctica on queda constància de la necessitat d'un bon disseny, d'unes bones interfícies entre capes i de les infinites possibilitats i característiques que li podem afegir. Algunes coses com el `FUSE` [1], un sistema de fitxers a nivell d'usuari, o les memòries cau plantejaven un repte molt interessant però la seva complexitat sobrepassa les expectatives de la pràctica. De tota manera queda clar podriem dedicar-li moltíssimes més hores, fer grups de quatre o sis persones i encara faltaria temps per implementar totes les parts que defineix el que és un sistema de fitxers modern.

[1] `http://fuse.sourceforge.net/`

A Codi font

A.1 Capa base

A.1.1 bitmap.h

```
1  /** Modul del mapa de bits. Aquesta part de les metadades s'encarrega de
2   * permetre un acces rapid a la localitzacio dels blocs de dades lliures. */
3
4  #include "common.h"
5
6  #define MAP_SIZE (BLOCK_SIZE*8)
7
8  typedef struct {
9        unsigned char valor[BLOCK_SIZE];
10 } emofs_bitmap;
11
12 /** Genera un mapa de zeros.
13  * @return: estructura de mapa de bits inicialitzat a zero.
14  */
15 emofs_bitmap map_of_zero();
16
17 /** Capgira tots els bits.
18  * Substitucio al mapa en parametre.
19  * @buf: el mapa de bits
20  */
21 void mflip(emofs_bitmap *buf);
22
23 /* Determina el primer bloc lliure d'un mapa de bits de sistema de fitxers.
24  * return: el nombre del bloc lliure. -1 si no s'en troben.
25  */
26 int find_first_free_block(void);
27
28 /** Imprimeix el mapa.
29  * @map: el mapa de bits
30  */
31 void print_map(emofs_bitmap *map);
32
33 /** Llegeix un bit. Cal recordar que sols pot ser un 1 o 0, per aixo es torna
34  * simplement amb un char. Si es -1 es perque hem intentat llegir un bit fora
35  * de rang.
36  * @bit: el nombre de bit a llegir
37  * @buf: el mapa de bits
```

```
38  * @return: el valor del bit. -1 si hi ha error de rang.
39  */
40 char mread(int bit, emofs_bitmap *buf);
41
42 /** Escriu un bit. Cal recordar que sols pot ser un 1 o 0, per aixo es torna
43  * simplement amb un char.
44  * @bit: el nombre de bit a llegir
45  * @val: el valor a desar.
46  * @buf: el mapa de bits
47  * @return: 0 si l'operacio funciona correctament.
48  */
49 char mwrite(int bit, unsigned char val, emofs_bitmap *buf);
50
51 /** Llegeix l'estat d'un bloc de dades representat al mapa de bits.
52  * @int block: nombre de block que buscam
53  * @return: 1 si esta ocupat.
54  */
55 char bm_read(int block);
56
57 /** Llegeix l'estat d'un bloc de dades representat al mapa de bits.
58  * @int block: nombre de block que buscam
59  * @return: 1 si esta ocupat.
60  */
61 int bm_write(int block, unsigned char val);
```

A.1.2 bitmap.c

```
1  #include <stdio.h>
2  #include "super.h"
3  #include "bitmap.h"
4
5  /** Genera un mapa de zeros.
6   * @return: estructura de mapa de bits inicialitzat a zero.
7   */
8  emofs_bitmap map_of_zero() {
9        int i = 0;
10       emofs_bitmap mapa;
11       for(i = 0; i < BLOCK_SIZE; i++) {
12             mapa.valor[i] = 0;
13       }
14       return mapa;
15 }
```

```
16
17 /** Capgira tots els bits.
18  * Substitucio al mapa en parametre.
19  * @buf: el mapa de bits
20  */
21 void mflip(emofs_bitmap *buf) {
22       int i = 0;
23       for(i = 0; i < BLOCK_SIZE; i++) {
24             buf->valor[i] = ~buf->valor[i];
25       }
26 }
27
28 /* Determina el primer bloc lliure d'un mapa de bits de sistema de fitxers.
29  * return: el nombre del bloc lliure. -1 si no s'en troben.
30  */
31 int find_first_free_block(void) {
32       int i, found, max;
33       emofs_superblock sb;
34       sbread(&sb);
35
36       if(sb.free_block_count == 0) {
37             printf("find_first_free_block: no hi ha blocs lliures\n");
38             return -2;
39       }
40
41       max = MAP_SIZE * (sb.last_bitm_block - sb.first_bitm_block +1);
42       found = 0;
43       i = 0;
44       while((i < max) && !found) {
45             if(bm_read(i) == 0) {
46                   found = 1;
47             } else {
48                   i++;
49             }
50       }
51
52       if (!found) {
53             printf("find_first_free_block: no s'ha trobat bloc\n");
54             return -1;
55       }
56
57       return i;
58 }
```

```
59
60 /** Imprimeix el mapa.
61  * @map: el mapa de bits
62  */
63 void print_map(emofs_bitmap *map) {
64       int i = 0;
65       for(i = 0; i < BLOCK_SIZE; i++) {
66             printf("%d", map->valor[i]);
67       }
68       printf("\n");
69 }
70
71 /** Llegeix un bit. Cal recordar que sols pot ser un 1 o 0, per aixo es torna
72  * simplement amb un char.
73  * Si es -1 es perque hem intentat llegir un bit fora de rang.
74  * @bit: el nombre de bit a llegir
75  * @buf: el mapa de bits
76  * @return: el valor del bit. -1 si hi ha error de rang.
77  */
78 char mread(int bit, emofs_bitmap *buf) {
79       unsigned char val = 0;
80       int grup = 0;
81       int posicio = 0;
82       if ((bit >= MAP_SIZE) || (bit < 0)) {
83             return -1;
84       }
85       grup = bit / 8;
86       posicio = bit % 8;
87
88       val = buf->valor[grup];
89       val &= (1 << posicio);
90       val >>= posicio;
91       return val;
92 }
93
94 /** Escriu un bit. Cal recordar que sols pot ser un 1 o 0, per aixo es torna
95  * simplement amb un char.
96  * @bit: el nombre de bit a llegir
97  * @val: el valor a desar.
98  * @buf: el mapa de bits
99  * @return: 0 si l'operacio funciona correctament.
100 */
101 char mwrite(int bit, unsigned char val, emofs_bitmap *buf) {
```

```
102         unsigned char tmp = 0;
103         unsigned int grup = 0;
104         unsigned int posicio = 0;
105 
106         grup = bit / 8;
107         posicio = bit % 8;
108         tmp = ~(1 << posicio);
109         val &= 1;
110         val <<= posicio;
111 
112         buf->valor[grup] &= tmp;
113         buf->valor[grup] |= val;
114         return 0;
115 }
116 
117 /** Llegeix l'estat d'un bloc de dades representat al mapa de bits.
118  * @int block: nombre de block que buscam
119  * @return: 1 si esta ocupat.
120  * 0 si esta lliure.
121  * -1 en cas d'error (fora de rang)
122  */
123 char bm_read(int block) {
124         emofs_bitmap bm;
125         emofs_superblock sb;
126         int n_bm, group, pos;
127 
128         sbread(&sb);
129 
130         /* Comprovam que no sigui fora de rang */
131         if (block < 0 || block > sb.total_data_block_count) {
132               puts("bm_read: acces fora de rang");
133               return -1;
134         }
135         group = block / MAP_SIZE;
136         pos = block % MAP_SIZE;
137         bread(sb.first_bitm_block + group, &bm);
138         return mread(pos, &bm);
139 }
140 
141 /** Escriu l'estat d'un bloc de dades representat al mapa de bits.
142  * @int block: nombre de block que buscam
143  * @return: error.
144  */
```

```
145 int bm_write(int block, unsigned char bit) {
146       int n_block, pos;
147       emofs_superblock sb;
148       emofs_bitmap bm;
149 
150       sbread(&sb);
151 
152       /* Comprovam que no sigui fora de rang */
153       if (block < 0 || block > sb.total_data_block_count) {
154             puts("bm_write: acces fora de rang");
155             return -1;
156       }
157       /* Calculam en quin block del mapa de bits es troba */
158       n_block = (block / MAP_SIZE) + sb.first_bitm_block;
159       pos = block % MAP_SIZE;
160       bread(n_block, &bm);
161       mwrite(pos, bit, &bm);
162       bwrite(n_block, &bm);
163       return 0;
164 }
```

A.1.3 block.h

```
1  /** Modul de gestio de blocs. Els blocs son la unitat minima del sistema de
2   * fitxers i consisteixen en un trocet de memoria la qual pot usar-se com
3   * inode, mapa de bits, bloc de dades o superbloc. */
4 
5  #include "common.h"
6 
7  /* Estructura de dades que representa un bloc de BLOCK_SIZE*bytes. */
8  typedef struct {
9       unsigned char valor[BLOCK_SIZE];
10 } emofs_block;
11 
12 /** Genera un bloc ple de zeros.
13  * @return: estructura de bloc de zeros
14  */
15 emofs_block block_of_zero();
16 
17 /** Genera un bloc ple d'uns.
18  * @return: estructura de bloc d'uns
19  */
```

```
20 emofs_block block_of_one();
21
22 /** Munta una imatge de sistema de fitxers.
23  * Crea el fitxer si no existeix.
24  * @return: el descriptor de fitxer
25  */
26 int bmount(void);
27
28 /** Desmunta la imatge en us.
29  * No te parametres perque el fitxer en us es desa a una variable
30  * estatica interna.
31  * @return: el codi d'error del close
32  */
33 int bumount(void);
34
35 /** Escriu el buffer al bloc assenyalat.
36  * @bloque: nombre de bloc a on escriure
37  * @buf: buffer d'entrada
38  * @return: sempre zero
39  */
40 int bwrite(int bloque, const void *buf);
41
42
43 /** Llegeix el bloc assenyalat i fica el contingut dins el buffer.
44  * @bloque: nombre de bloc d'on llegir
45  * @buf: buffer de sortida
46  * @return: sempre zero
47  */
48 int bread(int bloque, void *buf);
```

A.1.4 block.c

```
1  #include <errno.h>
2  #include <fcntl.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <stdlib.h>
6  #include <sys/stat.h>
7  #include <sys/types.h>
8  #include <unistd.h>
9
10 #include "block.h"
```

```
11
12 static int emofs_fs = 0;
13 static int mutex = 0;
14
15 emofs_block block_of_zero() {
16       emofs_block new_block;
17       memset(&new_block, 0x00, BLOCK_SIZE);
18       return new_block;
19 }
20
21 emofs_block block_of_one() {
22       emofs_block new_block;
23       memset(&new_block, 0xff, BLOCK_SIZE);
24       return new_block;
25 }
26
27 /** Munta la imatge del sistema de fitxers. Posa en marxa el semafor.
28  * @cami: cami de la imatge
29  * @return: 0 si exit
30  */
31 int bmount(void) {
32       emofs_fs = open(EMOFS_IMAGE_FILE, O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
33       if (emofs_fs == -1) {
34             printf("Error en crear el fitxer: %s\n", strerror(errno));
35       }
36       return emofs_fs;
37 }
38
39 int bumount(void) {
40       int error;
41       error = close(emofs_fs);
42       if (error == -1) {
43             printf("Error al tancar el fitxer: %s\n", strerror(errno));
44       }
45       return error;
46 }
47
48 int bwrite(int bloque, const void *buf) {
49       int posicio_bloc = bloque * BLOCK_SIZE;
50       int error = 0;
51       error = lseek(emofs_fs, posicio_bloc, SEEK_SET);
52       if (error == -1) {
53             printf("bwrite: error en lseek: %s\n", strerror(errno));
```

```
54              exit(-1);
55        }
56        error = write(emofs_fs, buf, BLOCK_SIZE);
57        if (error == -1) {
58              printf("Error en escriure al fitxer: %s\n", strerror(errno));
59              exit(-1);
60        }
61        return 0;
62  }
63
64  int bread(int bloque, void *buf) {
65        int posicio_bloc = bloque * BLOCK_SIZE;
66        int error = 0;
67        error = lseek(emofs_fs, posicio_bloc, SEEK_SET);
68        if (error == -1) {
69              printf("bread: error en lseek: %s\n", strerror(errno));
70              printf("posicio_bloc: %d\n", posicio_bloc);
71              exit(-1);
72        }
73        error = read(emofs_fs, buf, BLOCK_SIZE);
74        if (error == -1) {
75              printf("A la lectura: %s\n", strerror(errno));
76              exit(-1);
77        }
78        return 0;
79  }
```

A.1.5 block_lib.h

```
1  /** Biblioteca de gestio de blocs. Encapsula tasques com crear un mapa de bits,
2   * el vector d'inodes o reservar un inode. */
3
4  #include "common.h"
5
6  #include "super.h"
7
8  /* Decisions de disseny.
9   * - El primer bit del mapa de bits correspon al primer bloc, el del superbloc.
10  * - L'ordre sempre es sb - bm - ai
11  * - El valor "null" pels inodes correspon al zero.
12  */
13
```

```
14 /** Determina el nombre de blocs necessaris per situar el mapa de bits.
15  * @n_block: nombre de blocs (de dades)
16  * @return: nombre de blocs pel mapa de bits
17  */
18 int bitmap_size(int n_block);
19
20 /** Determina el nombre de blocs necessaris per contenir el vector d'inodes.
21  * @n_inode: nombre d'inodes que desitjam
22  * @return: nombre de blocs logics necessaris
23  */
24 int inode_array_size(int n_inode);
25
26 /** Inicialitza el superblock.
27  * D'aqui es fara el proces i calculs necessaris usant bitmap_size,
28  * inode_array_size
29  * @n_block: nombre total de blocs del sistema de fitxers
30  */
31 int init_superblock(int n_block);
32
33 /** Inicialitza el mapa de bits.
34  * Llegeix informacio del superblock, per tant ha d'estar inicialitzat.
35  * @return: 0 si exit.
36  */
37 int init_bitmap();
38
39 /** Inicialitza l'array d'inodes.
40  * Ha de fer la crida per
41  * @first_block: la posicio logica del primer inode
42  * @n_inode: nombre d'inodes a crear pel sistema de fitxers
43  */
44 int init_inode_array();
45
46 /** Reserva un bloc. S'encarrega de busca-lo, reservar-lo, deixar-lo marcat al
47  * mapa de bits i tornar-lo per poder treballar sobre ell. Modifica el nombre
48  * total de blocs del superbloc. Torna -1 si hi no havia blocs lliures.
49  * @type: Tipus de inode (directori o fitxer)
50  * @return: el numero del bloc reservat
51  */
52 int alloc_block();
53
54 /** Allibera un bloc. S'encarrega de tornar a posar a zero el seu bit del mapa
55  * de bits. Torna -1 si el bloc no estava ocupat. Modifica el nombre total de
56  * blocs del superbloc.
```

```
57  * @n_block: numero del bloc que volem alliberar.
58  * @return: error d'execucio.
59  */
60 int free_block(int n_block);
61
62 /** Reserva un inode. Modifica el nombre d'inodes lliures i totals del
63  * superbloc. Cal inicialitzar totes les dades de l'inode: tipus, tamany,
64  * mtime, recomptes i posar a zero els punters directes i indirectes. Modificar
65  * la llista enllacada d'inodes lliures.
66  * @return: numero d'inode, -1 si no en queden o es produeix un error.
67  */
68 int alloc_inode(int type);
69
70 /** Allibera un inode. Fa el recorregut de tots els inodes indirectes. Allibera
71  * tambe tots els blocs de dades. Modifica el nombre d'inodes lliures i totals
72  * del superbloc. Despres afegeix l'inode actual a la llista d'inodes lliures.
73  * @n_inode: numero de l'inode a lliberar.
74  * @return: -1 si el numero d'inode no era correcte.
75  */
76 int free_inode(int n_inode);
```

A.1.6 block_lib.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  #include "block.h"
6  #include "inode.h"
7  #include "bitmap.h"
8  #include "block_lib.h"
9
10 int bitmap_size(int n_block) {
11       int no_exacte;
12
13       no_exacte = ((n_block % MAP_SIZE) != 0);
14       return (n_block/MAP_SIZE) + no_exacte;
15 }
16
17 /** Calcul del nombre d'inodes. Permetrem tenir tants de inodes com blocs de
18  * dades aixi el nombre de fitxers petits i no buits sera maxim. Recodem que el
19  * tamany de l'inode esta triat per a que sigui un divisor del tamany del bloc
```

```
20  * per no tenir problemes */
21
22  int inode_array_size(int n_inode) {
23        int no_exacte, n_blocks;
24        if ((BLOCK_SIZE % sizeof(emofs_inode)) != 0) {
25              printf("Error en el calcul del tamany de l'array de inodes\n");
26              return -1;
27        }
28        if (n_inode*sizeof(emofs_inode) % BLOCK_SIZE) {
29              no_exacte = 1;
30        } else {
31              no_exacte = 0;
32        }
33        n_blocks = n_inode * sizeof(emofs_inode) / BLOCK_SIZE;
34        return n_blocks + no_exacte;
35  }
36
37  int init_superblock(int n_block) {
38        emofs_superblock sb;
39        int i;
40
41        /* El superbloc nomes ocupa el primer bloc */
42        sb.first_bitm_block = 1;
43        sb.last_bitm_block = sb.first_bitm_block + bitmap_size(n_block) - 1;
44        sb.first_inode_block = sb.last_bitm_block + 1;
45        /* Hem establert que tindem un inode per cada block */
46        sb.last_inode_block = sb.first_inode_block + \
47                        inode_array_size(n_block) - 1;
48        sb.first_data_block = sb.last_inode_block + 1;
49        /* Comencam a comptar a partir de 0 per tant -1 */
50        sb.last_data_block = n_block - 1;
51        sb.root_inode = 0;
52        sb.first_free_inode = 0;
53        sb.free_block_count = sb.last_data_block - sb.first_data_block + 1;
54        sb.free_inode_count = (1 + sb.last_inode_block - sb.first_inode_block) \
55                        * BLOCK_SIZE/sizeof(emofs_inode);
56        sb.total_block_count = n_block;
57        sb.total_inode_count = sb.free_inode_count;
58        sb.total_data_block_count = sb.last_data_block-sb.first_data_block+1;
59
60        for(i = 0; i < PADDING_BYTES; i++) {
61              sb.padding[i] = 'p';
62        }
```

```
63        return sbwrite(&sb);
64  }
65
66  /** Posar a zero tots els blocs lliures */
67  int init_bitmap() {
68        int i;
69        emofs_superblock sb;
70        emofs_block one;
71
72        one = block_of_one();
73
74        sbread(&sb);
75        for(i = sb.first_bitm_block; i <= sb.last_bitm_block; i++) {
76              bwrite(i, &one);
77        }
78        for(i = sb.first_data_block; i < sb.total_data_block_count; i++) {
79              bm_write(i, 0);
80        }
81        return 0;
82  }
83
84  int init_inode_array() {
85        int i;
86        emofs_inode inode;
87        emofs_superblock sb;
88
89        sbread(&sb);
90
91        /* Inicialitzam l'inode */
92        inode.type = FREE_INODE;
93        inode.size = 0;
94        inode.mtime = 0;
95        inode.block_count = 0;
96        inode.link_count = 0;
97        for(i = 0; i < DIRECT_POINTER_COUNT; i++) {
98              inode.direct_pointer[i] = 0;
99        }
100       for(i = 0; i < INDIRECT_POINTER_COUNT; i++) {
101             inode.indirect_pointer[i] = 0;
102       }
103
104       /* Enllacar inodes */
105       for (i = 0; i < (sb.total_inode_count - 1); i++) {
```

```
106             inode.size = i + 1;
107             write_inode(i, &inode);
108       }
109
110       /* Indicam que es el darrer inode */
111       inode.size = -1;
112       write_inode(i, &inode);
113       sb.first_free_inode = 0;
114       sbwrite(&sb);
115       return 0;
116 }
117
118
119 /** Reserva un bloc. S'encarrega de busca-lo, reservar-lo, deixar-lo marcat al
120  * mapa de bits i tornar-lo per poder treballar sobre ell. Modifica el nombre
121  * total de blocs del superbloc. Torna -1 si hi no havia blocs lliures.
122  * @return: el numero del bloc reservat
123  */
124 int alloc_block() {
125       int block;
126       emofs_superblock sb;
127       sbread(&sb);
128
129       if (sb.free_block_count == 0) {
130             printf("alloc_block: No hi ha blocs lliures\n");
131             return -1;
132       }
133
134       block = find_first_free_block();
135       if (block < 0) {
136             printf("alloc_block: No s'ha trobat bloc lliure\n");
137             return -2;
138       }
139
140       bm_write(block, 1);
141       sb.free_block_count--;
142       sbwrite(&sb);
143       return block;
144 }
145
146 int free_block(int n_block) {
147       emofs_superblock sb;
148       if (bm_read(n_block) == 0) {
```

```
149             puts("free_block: s'ha intentat alliberar un bloc lliure");
150             return -1;
151       }
152
153       sbread(&sb);
154       bm_write(n_block, 0);
155       sb.free_block_count++;
156       sbwrite(&sb);
157       return 0;
158 }
159
160 /* Totes les operacions d'inodes venen despres d'una lectura de bloc. Cal
161  * recordar que dins un mateix bloc tenim varis inodes i sols podem modificar
162  * amb el que esteim treballant.
163  */
164
165 /** Reserva un inode. Modifica el nombre d'inodes lliures i totals del
166  * superbloc. Cal inicialitzar totes les dades de l'inode: tipus, tamany,
167  * mtime, recomptes i posar a zero els punters directes i indirectes. Modificar
168  * la llista enllacada d'inodes lliures.
169  * @return: numero d'inode, -1 si no en queden o es produeix un error.
170  */
171 int alloc_inode(int type) {
172       emofs_superblock sb;
173       emofs_inode inode;
174       int i, n_inode, error;
175
176       if (!(type == DIRECTORY_INODE || type == FILE_INODE)) {
177             printf("alloc_inode: Incorrect inode type: %d\n", type);
178             return -2;
179       }
180
181       sbread(&sb);
182
183       if (sb.free_inode_count == 0 || sb.first_free_inode == -1) {
184             puts("alloc_inode: No free inodes left!");
185             return -1;
186       }
187
188       sb.free_inode_count--;
189       n_inode = sb.first_free_inode;
190
191       error = read_inode(n_inode, &inode);
```

```
192         if (error == -1) {
193               puts("alloc_inode: Error while reading first free inode");
194               sbwrite(&sb);
195               return -1;
196         }
197
198         if (inode.type != FREE_INODE) {
199               puts("alloc_inode: corrupted filesystem");
200               puts("alloc_inode: no free inode found in free inode list");
201               sbwrite(&sb);
202               return -1;
203         }
204
205         /* Hem definit que dins el camp size desam el üsegent inode lliure. Si
206          * fos final de llista deixeriem al SB un -1. D'aquesta manera
207          * detectariem que la llista es buida. Si el primer bloc lliure es el
208          * -1.
209          */
210         sb.first_free_inode = inode.size;
211         inode.type = type;
212         inode.size = 0;
213         inode.mtime = time(NULL);
214         inode.block_count = 0;
215         inode.link_count = 0;
216
217         for(i = 0; i < DIRECT_POINTER_COUNT; i++) {
218               inode.direct_pointer[i] = NULL_POINTER;
219         }
220         for(i = 0; i < INDIRECT_POINTER_COUNT; i++) {
221               inode.indirect_pointer[i] = NULL_POINTER;
222         }
223
224         write_inode(n_inode, &inode);
225         sbwrite(&sb);
226
227         return n_inode;
228
229   }
230
231   int free_indirect_inode(int n_indirect, int level, emofs_superblock *sb) {
232         int i;
233         int indirect[INDIRECT_POINTERS_PER_BLOCK];
234         bread(n_indirect, &indirect);
```

```
235
236         if (level == 0) {
237               puts("free_indirect_inode: Us incorrecte");
238         } else if (level == 1) {
239               for (i = 0; i < INDIRECT_POINTERS_PER_BLOCK; i++) {
240                     if (indirect[i] != NULL_POINTER) {
241                           bm_write(indirect[i], 0);
242                           sb->free_block_count++;
243                     }
244               }
245               bm_write(n_indirect, 0);
246               sb->free_block_count++;
247         } else {
248               for (i = 0; i < INDIRECT_POINTERS_PER_BLOCK; i++) {
249                     if (indirect[i] != NULL_POINTER) {
250                           free_indirect_inode(indirect[i], level-1, sb);
251                     }
252               }
253         }
254
255         return 0;
256   }
257
258   /** Allibera un inode. Fa el recorregut de tots els inodes indirectes. Allibera
259    * tambe tots els blocs de dades. Modifica el nombre d'inodes i blocs lliures
260    * del superbloc. Despres afegeix l'inode actual a la llista d'inodes lliures.
261    * @n_inode: numero de l'inode a lliberar.
262    * @return: -1 si el numero d'inode no era correcte.
263    */
264   int free_inode(int n_inode) {
265         emofs_superblock sb;
266         emofs_inode inode;
267         int *pointers = malloc((INDIRECT_POINTER_COUNT+1)*sizeof(int));
268         int i, j, k, error;
269
270         sbread(&sb);
271         error = read_inode(n_inode, &inode);
272         if (error < 0) {
273               puts("free_inode: no s'ha pogut llegir inode");
274               return -1;
275         }
276
277         if (inode.type == FREE_INODE) {
```

```
278             puts("free_niode: s'ha intentat alliberar-ne un de lliure");
279             return -2;
280       }
281
282       for (i = 0; i < DIRECT_POINTER_COUNT; i++) {
283             if (inode.direct_pointer[i] != NULL_POINTER) {
284                   printf("free_inode: alliberam bloc %d\n", \
285                         inode.direct_pointer[i]);
286                   bm_write(inode.direct_pointer[i], 0);
287                   sb.free_block_count++;
288             }
289       }
290
291       for (i = 0; i < INDIRECT_POINTER_COUNT; i++) {
292             if (inode.indirect_pointer[i] != NULL_POINTER) {
293                   free_indirect_inode(inode.indirect_pointer[i], i+1, &sb);
294             }
295       }
296
297       /* Situam com a lliure i enllacam la llista */
298       inode.type = FREE_INODE;
299       inode.size = sb.first_free_inode;
300       write_inode(n_inode, &inode);
301
302       /* Establim l'inode com el primer lliure */
303       sb.first_free_inode = n_inode;
304       sb.free_inode_count++;
305
306       sbwrite(&sb);
307       return 0;
308
309 }
```

A.1.7 common.h

```
1  /* Modul de definicio de parametres comuns de tot el sistema de fitxers. */
2
3  /* Definicio dels parametres comuns al llarg de la practica. */
4  #define __u32 unsigned int
5  #define __u16 unsigned short
6
7  /* Un bloc te 4KB, com a minim pot ser de 512 Bytes (segons l'enunciat) */
```

```
8  #define BLOCK_SIZE 4096
9
10 /* Bloc on desam el superblock. */
11 #define SUPERBLOCK_BLOCK_N 0
12
13 #define NULL_POINTER (-1)
14
15 /* 60 perque 60B+4B d'un int fan 64B. D'aquesta manera podem fer caure be els
16  * dins els blocs. */
17 #define MAX_FILENAME_LEN 60
18
19 #define MAX_PATH_LEN 255
20
21 #define EMOFS_IMAGE_FILE "emo.fs"
22
23 /* Definicions d'acces als inodes. */
24 #define FREE_INODE 0
25 #define DIRECTORY_INODE 1
26 #define FILE_INODE 2
27
28 #define BUFFER_SIZE 255
```

A.2 Nucli

A.2.1 super.h

```
1  /** Modul del superbloc. Al superbloc desam informacio com quins tamanys tenen
2   * el mapa de bits, els inodes i les dades; quin es el primer inode lliure o
3   * quants de blocs de dades queden. */
4
5  #include "common.h"
6
7  /* Importantissim contar be el nombre d'items. */
8  #define SUPERBLOCK_ITEMS 13
9  #define SUPERBLOCK_SIZE (SUPERBLOCK_ITEMS*4)
10 #define PADDING_BYTES (BLOCK_SIZE-SUPERBLOCK_SIZE)
11
12 /* Como minimo, para SUPERBLOQUE:
13  * Numero del primer bloque del mapa de bits
14  * Numero del ultimo bloque del mapa de bits
15  * Numero del primer bloque del array de inodos
16  * Numero del ultimo bloque del array de inodos
```

```
17  * Numero del primer bloque de datos
18  * Numero del ultimo bloque de datos
19  * Numero del inodo del directorio raiz
20  * Numero del primer inodo libre
21  * Cantidad de bloques libres
22  * Cantidad de inodos libres
23  * Cantidad total de bloques
24  * Cantidad total de inodos */
25 typedef struct {
26       __u32 first_bitm_block;
27       __u32 last_bitm_block;
28       __u32 first_inode_block;
29       __u32 last_inode_block;
30       __u32 first_data_block;
31       __u32 last_data_block;
32       __u32 root_inode;
33       __u32 first_free_inode;
34       __u32 free_block_count;
35       __u32 free_inode_count;
36       __u32 total_block_count;
37       __u32 total_inode_count;
38       __u32 total_data_block_count;
39       unsigned char padding[PADDING_BYTES];
40 } emofs_superblock;
41
42 /** Escriu al superblock
43  * @sb: Contingut del superblock
44  * @return: -1 per error.
45  */
46 int sbread(emofs_superblock *sb);
47
48 /** Escriu el superblock.
49  * @sb: Contingut que ha de tenir el superblock
50  * @return: -1 per error
51  */
52 int sbwrite(emofs_superblock *sb);
53
54 /** Imprimeix el contigut del superbloc.
55  * @sb: el superblock.
56  */
57 void print_sb(emofs_superblock *sb);
```

A.2.2 super.c

```
1  #include <stdio.h>
2  #include "super.h"
3
4  /** Escriu al superblock.
5   * @sb: Contingut del superblock
6   * @return: error si n'hi ha.
7   */
8  int sbwrite(emofs_superblock *sb) {
9        return bwrite(SUPERBLOCK_BLOCK_N, sb);
10 }
11
12 /** Llegeix el superblock.
13  * @sb: Contingut del superblock
14  * @return: error si n'hi ha.
15  */
16 int sbread(emofs_superblock *sb) {
17       return bread(SUPERBLOCK_BLOCK_N, sb);
18 }
19
20 /** Imprimeix el contigut del superbloc.
21  * @sb: el superblock.
22  */
23 void print_sb(emofs_superblock *sb) {
24       printf("first_bitm_block %d \n", sb->first_bitm_block);
25       printf("sb.last_bitm_block %d \n", sb->last_bitm_block);
26       printf("sb.first_inode_block %d \n", sb->first_inode_block);
27       printf("sb.last_inode_block %d \n", sb->last_inode_block);
28       printf("sb.first_data_block %d \n", sb->first_data_block);
29       printf("sb.last_data_block %d \n", sb->last_data_block);
30       printf("sb.root_inode %d \n", sb->root_inode);
31       printf("sb.first_free_inode %d \n", sb->first_free_inode);
32       printf("sb.free_block_count %d \n", sb->free_block_count);
33       printf("sb.free_inode_count %d \n", sb->free_inode_count);
34       printf("sb.total_block_count %d \n", sb->total_block_count);
35       printf("sb.total_inode_count %d \n", sb->total_inode_count);
36       printf("sb.total_data_block_count %d\n", sb->total_data_block_count);
37 }
```

A.2.3 inode.h

```
1  /** Modul de creacio i gestio dels inodes. Els inodes son part de les metadades
2   * del sistema de fitxers i contenen informacio relativa a cada fitxer o
3   * directori. Indiquen la data de modificacio, el seu tamany o la quantitat
4   * d'enllacos que tenen. */
5
6  #include "common.h"
7
8  #define INODES_PER_BLOCK (BLOCK_SIZE/sizeof(emofs_inode))
9  #define INDIRECT_POINTERS_PER_BLOCK (BLOCK_SIZE/sizeof(int))
10 #define LEVEL_1_POINTERS (INDIRECT_POINTERS_PER_BLOCK)
11 #define LEVEL_2_POINTERS (LEVEL_1_POINTERS*LEVEL_1_POINTERS)
12 #define LEVEL_3_POINTERS (LEVEL_2_POINTERS*LEVEL_1_POINTERS)
13 #define OFFSET_L0_POINTERS (0)
14 #define OFFSET_L1_POINTERS (OFFSET_L0_POINTERS + DIRECT_POINTER_COUNT)
15 #define OFFSET_L2_POINTERS (OFFSET_L1_POINTERS + LEVEL_1_POINTERS)
16 #define OFFSET_L3_POINTERS (OFFSET_L2_POINTERS + LEVEL_2_POINTERS)
17
18 /* Tamany objectiu de l'inode. En bytes. */
19 #define INODE_TARGET_SIZE 64
20
21 /* Dades descriptores de l'inode. */
22 #define INODE_ITEMS 5
23 #define INDIRECT_POINTER_COUNT 3
24
25 /* Nombre de chars ocupats fins al moment */
26 #define INODE_MUST_SIZE ((INODE_ITEMS+INDIRECT_POINTER_COUNT)*4)
27
28 /* 8 per aque ens quadrin els blocks de 64bytes */
29 #define DIRECT_POINTER_COUNT ((INODE_TARGET_SIZE-INODE_MUST_SIZE)/4)
30
31 /* Como minimo, para INODO:
32  * Tipo (libre, directorio o fichero)
33  * Tamano en bytes logicos
34  * Fecha de modificacion
35  * Cantidad de bloques fisicos asignados
36  * Cantidad de enlaces fisicos
37  * Varios punteros a bloques directos (segun el tamano que queramos que tenga
38  * el inodo: se recomienda un tamano de 64 bytes)
39  * 3 punteros a bloques indirectos */
40 typedef struct {
41       /* 0 lliure, 1 directori, 2 fitxer, 3 buit */
42       __u32 type;
43       /* Si es lliure size indica la posicio del üsegent bloc lliure.
```

```
44          * -1 indica final de llista.*/
45         __u32 size;
46         __u32 mtime;
47         __u32 block_count;
48         __u32 link_count;
49         __u32 direct_pointer[DIRECT_POINTER_COUNT];
50         /* 0 es el de nivell 1, 1 el de nivell 2 i 2 el de nivell 3 */
51         __u32 indirect_pointer[INDIRECT_POINTER_COUNT];
52 } emofs_inode;
53
54 /* Localitzacio d'una dada adins un inode.
55  * n -> nivell de punter indirecte
56  * pex: 3 -> [0, [3, 0, 0]]
57  * pex: 10 -> [1, [2, 0, 0]]
58  */
59 typedef struct {
60         int n; /* de 0 a 3 */
61         int l[3];
62 } emofs_data_loc;
63
64 /** Situa dins l'estructura de localitzacio un numero de bloc de dades.
65  * @n_block: nombre sobre el qual fer el calcul.
66  * @loc: l'estructura de posicionament.
67  * @return: 0 si exit.
68  */
69 int localize_data(int n_block, emofs_data_loc *loc);
70
71 /** Escriu un inode.
72  * @n_inode: el nombre de l'inode a escriure.
73  * @buf: l'inode d'entrada.
74  * @return: -1 si l'inode objectiu no existeix.
75  */
76 int write_inode(int n_inode, emofs_inode *buf);
77
78 /** Escriu un inode.
79  * @n_inode: el nombre de l'inode a escriure.
80  * @buf: l'inode d'entrada.
81  * @date: hora de modificacio de l'inode.
82  * @return: -1 si l'inode objectiu no existeix.
83  */
84 int write_inode_time(int n_inode, emofs_inode *buf, time_t date);
85
86 /** Llegeix un inode.
```

```
87  * @n_inode: el nombre de l'inode a llegir.
88  * @buf: buffer on es desa el contingut de l'inode.
89  * @return: -1 si el nombre d'inode no existia.
90  */
91 int read_inode(int n_inode, emofs_inode *buf);
92
93 /** Tradueix el bloc logic al seu valor de bloc fisic.
94  * Cal recordar que si afegim un bloc de dades hem de modificar el contador
95  * d'inodes i corregir el tamany de fitxer.
96  * @n_inode: nombre d'inode
97  * @l_block: numero de bloc logic.
98  * @alloc: si 1 i no existeix el bloc fisic reserva un bloc de dades.
99  * @return: 0 si exit. -1 si es volia llegir i no existia.
100 */
101 int translate_inode(int n_inode, int l_block, int *block, int alloc);
```

A.2.4 inode.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "inode.h"
5  #include "block.h"
6  #include "super.h"
7
8  /* Totes les operacions d'inodes venen despres d'una lectura de bloc. Cal
9   * recordar que dins un mateix bloc tenim varis inodes i sols podem modificar
10  * amb el que esteim treballant.
11  */
12
13 /** Escriu un inode.
14  * @n_inode: el nombre de l'inode a escriure.
15  * @buf: l'inode d'entrada.
16  * @date: hora de modificacio de l'inode.
17  * @return: -1 si l'inode objectiu no existeix.
18  */
19 int write_inode_time(int n_inode, emofs_inode *buf, time_t date) {
20       int inode_block, inode_pos;
21       int error;
22       emofs_superblock sb;
23       emofs_inode inode_array[INODES_PER_BLOCK];
24
```

```
25        sbread(&sb);
26        inode_block = sb.first_inode_block + (n_inode/INODES_PER_BLOCK);
27        inode_pos = n_inode % INODES_PER_BLOCK;
28        error = bread(inode_block, inode_array);
29        memcpy(&buf->mtime, &date, sizeof(time_t));
30        memcpy(&inode_array[inode_pos], buf, sizeof(emofs_inode));
31        bwrite(inode_block, inode_array);
32        return error;
33  }
34
35  /** Escriu un inode.
36   * @n_inode: el nombre de l'inode a escriure.
37   * @buf: l'inode d'entrada.
38   * @return: -1 si l'inode objectiu no existeix.
39   */
40  int write_inode(int n_inode, emofs_inode *buf) {
41        return write_inode_time(n_inode, buf, time(NULL));
42  }
43
44  /** Llegeix un inode.
45   * @n_inode: el nombre de l'inode a llegir.
46   * @buf: buffer on es desa el contingut de l'inode.
47   * @return: -1 si el nombre d'inode no existia.
48   */
49  int read_inode(int n_inode, emofs_inode *buf) {
50        int inode_block = -1;
51        int inode_pos = -1;
52        int error = 0;
53        emofs_superblock sb;
54        emofs_inode inode_array[INODES_PER_BLOCK];
55
56        if (n_inode < 0) {
57              printf("read_inode: inode negatiu\n");
58              return -2;
59        }
60
61        error = sbread(&sb);
62        inode_block = sb.first_inode_block + (n_inode / INODES_PER_BLOCK);
63        inode_pos = n_inode % INODES_PER_BLOCK;
64
65        if (inode_block < 0) {
66              printf("read_inode: bloc d'inode negatiu\n");
67              return -3;
```

```
68        } else if (inode_pos < 0) {
69              printf("read_inode: posicio d'inode negatiu\n");
70              return -4;
71        }
72
73        bread(inode_block, inode_array);
74        memcpy(buf, &inode_array[inode_pos], sizeof(emofs_inode));
75
76        return error;
77  }
78
79  /** Situa dins l'estructura de localitzacio un numero de bloc de dades.
80   * @n_block: nombre sobre el qual fer el calcul.
81   * @loc: l'estructura de posicionament.
82   * @return: 0 si exit.
83   */
84  int localize_data(int n_block, emofs_data_loc *loc) {
85        int level, found;
86        int max[INDIRECT_POINTER_COUNT+1];
87        int group, pos;
88        /* Inicialitzacio del vector de maxims.
89         * maxim nivell = punters directes + \Sum indirectes^nivell
90         */
91        max[0] = DIRECT_POINTER_COUNT;
92        max[1] = LEVEL_1_POINTERS + max[0];
93        max[2] = LEVEL_2_POINTERS + max[1];
94        max[3] = LEVEL_3_POINTERS + max[2];
95
96        /* Busqueda del nivell. */
97        found = 0;
98        level = 0;
99        while ((!found) && (level < INDIRECT_POINTER_COUNT+1)) {
100             if (n_block < max[level]) {
101                   found = 1;
102             } else {
103                   level++;
104             }
105       }
106
107       if(!found) {
108             puts("Peticio d'un bloc fora de rang");
109             return -1;
110       }
```

```
111
112         loc->n = level;
113         switch(level) {
114         case 0:
115               loc->l[0] = n_block;
116               break;
117         case 1:
118               loc->l[0] = n_block - max[0];
119               break;
120         case 2:
121               n_block -= max[1];
122               loc->l[1] = n_block / LEVEL_2_POINTERS;
123               loc->l[0] = n_block % LEVEL_2_POINTERS;
124               break;
125         case 3:
126               n_block -= max[2];
127               n_block /= LEVEL_2_POINTERS;
128               loc->l[2] = n_block;
129               loc->l[1] = n_block / LEVEL_2_POINTERS;
130               loc->l[0] = n_block % LEVEL_2_POINTERS;
131               break;
132         default:
133               puts("Nombre de nivells incorrecte");
134               return -2;
135         }
136
137         return 0;
138   }
139
140   /** Inicialitza un bloc de punters a NULL_POINTER. Funcio interna. */
141   int init_ind_pointers(int ptrs[]) {
142         int i;
143
144         for (i=0; i < INDIRECT_POINTERS_PER_BLOCK; i++) {
145               ptrs[i] = NULL_POINTER;
146         }
147
148         return 0;
149   }
150
151   /*
152         bread(n_block, &i_block);
153         data = i_block[loc->l[n]];
```

```
154         if ((data == NULL_POINTER) && alloc) {
155               data = alloc_block();
156               if (data < 0) {
157                     return -2;
158               }
159               init_ind_pointers(i_block);
160               i_block[loc->l[n]] = data;
161               bwrite(n_block, &i_block);
162               inode.block_count++;
163               write_inode(n_inode, &inode);
164         }
165   */
166
167   int translate_indirect_1(int *block, emofs_inode *inode,
168                         emofs_data_loc *loc, int alloc);
169   int translate_indirect_2(int *block, emofs_inode *inode,
170                         emofs_data_loc *loc, int alloc);
171   int translate_indirect_3(int *block, emofs_inode *inode,
172                         emofs_data_loc *loc, int alloc);
173
174
175   /** Busca i reserva un nou bloc de dades dins els punters directes.
176    * @data: el nombre del bloc de dades
177    * @inode: punter a l'inode amb que treballam
178    * @pos: posicio dins el vector directe
179    * @alloc: si cal reservar en cas que estigui lliure
180    * @return: 0 si exit,
181    * -2 si no hi ha espai,
182    * -1 si es vol llegir sense existir el bloc
183    */
184   int translate_inode_indirect(int *block, emofs_inode *inode,
185                           emofs_data_loc *loc, int alloc) {
186         int error;
187
188         switch(loc->n) {
189         case 1: error = translate_indirect_1(block, inode, loc, alloc); break;
190         case 2: error = translate_indirect_2(block, inode, loc, alloc); break;
191         case 3: error = translate_indirect_3(block, inode, loc, alloc); break;
192         default:
193               puts("traduir_indirecte: Nivells incorrectes");
194               error = -3;
195               break;
196         }
```

```
197       return error;
198 }
199
200 int translate_indirect_1(int *block, emofs_inode *inode,
201                   emofs_data_loc *loc, int alloc){
202       int n_indirect = inode->indirect_pointer[0];
203       int data_indirect[INDIRECT_POINTERS_PER_BLOCK];
204       if(n_indirect == NULL_POINTER) {
205             if (alloc) {
206                   n_indirect = alloc_block();
207                   inode->indirect_pointer[0] = n_indirect;
208                   /* Inicialitzam el block de punters */
209                   init_ind_pointers(data_indirect);
210                   bwrite(n_indirect, data_indirect);
211                   inode->block_count++;
212             } else{
213                   return -1;
214             }
215       }
216       bread(n_indirect, data_indirect);
217       *block = data_indirect[loc->l[0]];
218       if (*block == NULL_POINTER) {
219             if (alloc) {
220                   *block = alloc_block();
221                   if (*block < 0) {
222                         return -2;
223                   }
224             } else {
225                   return -1;
226             }
227             data_indirect[loc->l[0]] = *block;
228             bwrite(n_indirect, data_indirect);
229             inode->block_count++;
230       }
231       return 0;
232 }
233
234 int translate_indirect_2(int *block, emofs_inode *inode,
235                   emofs_data_loc *loc, int alloc){
236       int n_indirect2, n_indirect1;
237       int data_indirect2[INDIRECT_POINTERS_PER_BLOCK];
238       int data_indirect1[INDIRECT_POINTERS_PER_BLOCK];
239
```

```
240         n_indirect2 = inode->indirect_pointer[1];
241         if (n_indirect2 == NULL_POINTER) {
242               if (alloc) {
243                     n_indirect2 = alloc_block();
244                     inode->indirect_pointer[1] = n_indirect2;
245                     /* Inicialitzam el block de punters a punters */
246                     init_ind_pointers(data_indirect2);
247                     bwrite(n_indirect2, data_indirect2);
248                     inode->block_count++;
249               } else {
250                     return -1;
251               }
252         }
253         bread(n_indirect2, data_indirect2);
254         n_indirect1 = data_indirect2[loc->l[1]];
255         if (n_indirect1 == NULL_POINTER) {
256               if (alloc) {
257                     n_indirect1 = alloc_block();
258                     /* Inicialitzam el block de punters a dades */
259                     init_ind_pointers(data_indirect1);
260                     bwrite(n_indirect1, data_indirect1);
261                     data_indirect2[loc->l[1]] = n_indirect1;
262                     bwrite(n_indirect2, data_indirect2);
263                     inode->block_count++;
264               } else {
265                     return -1;
266               }
267         }
268         bread(n_indirect1, data_indirect1);
269         *block = data_indirect1[loc->l[0]];
270         if (*block == NULL_POINTER) {
271               if (alloc) {
272                     *block = alloc_block();
273                     if (*block < 0) {
274                           return -2;
275                     }
276               } else {
277                     return -1;
278               }
279               data_indirect1[loc->l[0]] = *block;
280               bwrite(n_indirect1, data_indirect1);
281               inode->block_count++;
282         }
```

```
283         return 0;
284 }
285
286
287 int translate_indirect_3(int *block, emofs_inode *inode,
288                     emofs_data_loc *loc, int alloc) {
289         int n_indirect3, n_indirect2, n_indirect1;
290         int data_indirect3[INDIRECT_POINTERS_PER_BLOCK];
291         int data_indirect2[INDIRECT_POINTERS_PER_BLOCK];
292         int data_indirect1[INDIRECT_POINTERS_PER_BLOCK];
293
294         n_indirect3 = inode->indirect_pointer[2];
295         if(n_indirect3 == NULL_POINTER) {
296               if (alloc) {
297                     n_indirect3 = alloc_block();
298                     inode->indirect_pointer[2] = n_indirect3;
299                     /* Inicialitzam el block de punters a punters a punters */
300                     init_ind_pointers(data_indirect3);
301                     bwrite(n_indirect3, data_indirect3);
302                     inode->block_count++;
303               } else {
304                     return -1;
305               }
306         }
307         bread(n_indirect3, data_indirect3);
308         n_indirect2 = data_indirect3[loc->l[1]];
309         /* Segon nivell */
310         if (n_indirect2 == NULL_POINTER) {
311               if (alloc) {
312                     n_indirect2 = alloc_block();
313                     data_indirect3[loc->l[2]] = n_indirect2;
314                     bwrite(n_indirect3, data_indirect3);
315                     /* Inicialitzam el block de punters a punters a dades */
316                     init_ind_pointers(data_indirect2);
317                     bwrite(n_indirect2, data_indirect2);
318                     inode->block_count++;
319               } else {
320                     return -1;
321               }
322         }
323         bread(n_indirect2, data_indirect2);
324         n_indirect1 = data_indirect2[loc->l[1]];
325         /* Primer nivell */
```

```
326         if(n_indirect1 == NULL_POINTER) {
327               if (alloc) {
328                     n_indirect1 = alloc_block();
329                     data_indirect2[loc->l[1]] = n_indirect1;
330                     bwrite(n_indirect2, data_indirect2);
331                     /* Inicialitzam el block de punters a dades*/
332                     init_ind_pointers(data_indirect1);
333                     bwrite(n_indirect1, data_indirect1);
334                     inode->block_count++;
335               } else {
336                     return -1;
337               }
338         }
339         bread(n_indirect1, data_indirect1);
340         *block = data_indirect1[loc->l[0]];
341         if (*block == NULL_POINTER) {
342               if (alloc) {
343                     *block = alloc_block();
344                     if (*block < 0) {
345                           return -2;
346                     }
347               } else {
348                     return -1;
349               }
350               data_indirect1[loc->l[0]] = *block;
351               bwrite(n_indirect1, data_indirect1);
352               inode->block_count++;
353         }
354         return 0;
355 }
356
357 /** Busca i reserva un nou bloc de dades dins els punters directes.
358  * @data: el nombre del bloc de dades
359  * @inode: punter a l'inode amb que treballam
360  * @pos: posicio dins el vector directe
361  * @alloc: si cal reservar en cas que estigui lliure
362  * @return: 0 si exit,
363  * -2 si no hi ha espai,
364  * -1 si es vol llegir sense existir el bloc
365  */
366 int translate_inode_direct(int *block, emofs_inode *inode, int pos, int alloc) {
367         *block = inode->direct_pointer[pos];
368         if (*block == NULL_POINTER) {
```

```
369             if (alloc) {
370                   *block = alloc_block();
371                   if (*block < 0) {
372                         return -2;
373                   }
374                   inode->direct_pointer[pos] = *block;
375                   inode->block_count++;
376             } else{
377                   return -1;
378             }
379       }
380       return 0;
381 }
382
383
384 /** Tradueix el bloc logic al seu valor de bloc fisic.
385  * Cal recordar que si afegim un bloc de dades hem de modificar el contador
386  * d'inodes pero no corregim el tamany de fitxer.
387  * @n_inode: nombre d'inode
388  * @l_block: numero de bloc logic.
389  * @block: numero de bloc fisic
390  * @alloc: si 1 i no existeix el bloc fisic reserva un bloc de dades.
391  * @return: 0 si exit. -1 si es volia llegir i no existia. -2 en cas de
392  * problemes en la solicitud de blocs (escriptura).
393  */
394 int translate_inode(int n_inode, int l_block, int *block, int alloc) {
395       int level, i, n_block;
396       emofs_inode inode;
397       int i_block[INDIRECT_POINTERS_PER_BLOCK];
398       emofs_data_loc loc;
399       int error;
400
401       error = read_inode(n_inode, &inode);
402       if (error < 0) {
403             puts("No s'ha trobat l'inode");
404             return error;
405       }
406       localize_data(l_block, &loc);
407       if (loc.n == 0) {
408             error = translate_inode_direct(block, &inode, loc.l[0], alloc);
409       } else {
410             error = translate_inode_indirect(block, &inode, &loc, alloc);
411       }
```

```
412       write_inode(n_inode, &inode);
413       return error;
414 }
```

A.2.5 file.h

```
1  /* Modul dels fitxers. Els fitxers son una abstraccio per treballar sobre els
2   * inodes. Alhora permeten manipular les dades de l'usuari amb un cami de
3   * fitxer (/home/usuari/.signature) enlloc de amb el nombre d'inode. */
4
5  #include "common.h"
6
7  #include <time.h>
8
9  /* L'estructura emofs_stat es exacta a la de l'inode excepte que no desa punters.
10  * Consultar inode.h per mes informacio.
11  */
12 typedef struct {
13       __u32 type;
14       __u32 size;
15       __u32 mtime;
16       __u32 block_count;
17       __u32 link_count;
18 } emofs_inode_stat;
19
20 /** Escriu un cert nombre de bytes d'un fitxer. Cal recordar que si indicam un
21  * offset = 10 significa que hem de comencar. Es important actualitzar el
22  * tamany del fitxer a escriure des del byte numero 10.
23  * @inode: nombre d'inode on volem escriure
24  * @buffer: buffer d'entrada. D'alla copiarem les dades.
25  * @offset: determinam el primer byte del fitxer.
26  * @n_bytes: nombre de bytes a escriure
27  * @return: el nombre de bytes escrits.
28  */
29 int write_file(int inode, const void *buffer, int offset, int n_bytes);
30
31 /** Llegeix un cert nombre de bytes d'un fitxer.
32  * Cal recordar que si indicam un offset = 10 significa que hem de comencar.
33  * Es important actualitzar el tamany del fitxer.
34  * a escriure des del byte numero 10.
35  * @inode: nombre d'inode on volem escriure
36  * @buffer: Punter buffer de sortida, aquesta funcio reservara l'espai
```

```
37  * @offset: determinam el primer byte del fitxer.
38  * @n_bytes: nombre de bytes a escriure
39  * @return: el nombre de bytes escrits.
40  */
41 int read_file(int inode, void **buf, int offset, int n_bytes);
42
43 /** Trunca un fitxer a partir del byte n.
44  * Si n_bytes = 0 alliberam tots els blocs.
45  * @inode: el nombre d'inode.
46  * @n_byte: el byte final del fitxer.
47  * @return: 0 si exit.
48  */
49 int truncate_file(int inode, int n_byte);
50
51 /** Obte la informacio d'un inode.
52  * @inode: nombre del fitxer.
53  * @stat: el punter de sortida de les dades.
54  * @return: 0 si exit.
55  */
56 int stat_file(int inode, emofs_inode_stat *stat);
57
58 /** Canvi el la data de modificacio de l'inode.
59  * @inode: nombre del fitxer
60  * @date: data epoch a posar
61  * @return: 0 si exit
62  */
63 int timestamp_file(int inode, time_t date);
```

A.2.6 file.c

```
1  #include <string.h>
2  #include <stdlib.h>
3
4  #include "file.h"
5  #include "inode.h"
6  #include "block.h"
7  #include "block_lib.h"
8
9  /*
10 Calculam el bloc a que pertany el primer byte a llegir.
11 Llegim l'inode.
12 Tres casos:
```

```
13        - acabar el primer bloc.
14        - llegir completament els blocs intermitjos.
15        - comencar el darrer bloc.
16  D'aquesta manera es bona idea calcular previament a quin cas
17  pertany.
18  */
19
20  /** Funcio interna per calcular els blocs necessaris per la tasca.
21   * Recorda que BLOCK_SIZE determina el nombre de bytes per bloc.
22   * @offset: desplacament inicial.
23   * @n_bytes: bytes a llegir.
24   * @first: nombre de bloc del primer de tots.
25   * @f_offset: desplacament dins el primer bloc.
26   * @l_size: tamany del darrer bloc si n'hi ha mes d'un.
27   * @return:  1 sols primer
28   *  2 primer i segon
29   *  3+ un de comencament, final i alguns intermijos
30   */
31  int calc_blocks_with_offset(int offset, int n_bytes,
32                        int *first, int *f_offset, int *l_size) {
33        int how_many = 0;
34        *first = offset / BLOCK_SIZE;
35        *f_offset = offset % BLOCK_SIZE;
36        *l_size = ((*f_offset) + n_bytes)%BLOCK_SIZE;
37        how_many = (*f_offset + n_bytes)/BLOCK_SIZE;
38        how_many += ((*f_offset + n_bytes) % BLOCK_SIZE) ? 1 : 0;
39        return how_many;
40
41  }
42
43  /** Escriu un cert nombre de bits d'un fitxer. Cal recordar que si indicam un
44   * offset = 10 significa que hem de comencar per el byte nombre 10 del fitxer.
45   * Es important actualitzar el tamany del fitxer. a escriure des del byte
46   * numero 10.
47   * @inode: nombre d'inode on volem escriure
48   * @buffer: buffer d'entrada. D'alla copiarem les dades.
49   * @offset: determinam el primer byte del fitxer.
50   * @n_bytes: nombre de bytes a escriure
51   * @return: el nombre de bytes escrits.
52   */
53  int write_file(int inode, const void *buffer, int offset, int n_bytes) {
54        int error, i;
55        /* quants de blocs tenim a llegir, nombre de bloc logic a treballar */
```

```
56        int how_many = 0;
57        int block = 0;
58        /* Nombre de blocs llegits, necessari pels offsets del memcpy. */
59        int n_blocks_written = 0;
60        int n_bytes_written = 0;
61        int n_block = 0; /* nombre del primer bloc */
62        int f_offset = 0; /* offset del primer bloc */
63        int l_size = 0; /* tamany al darrer bloc */
64        int bytes_copy = 0; /* Bytes a escriure a la fase */
65        int bytes_offset = 0; /* Bytes a desplacar-se */
66        int grow_size = 0;
67        emofs_block tmp_block;
68        emofs_inode tmp_inode;
69        /* recorda: offset, bytes a llegir, primer block, offset del primer,
70         * final del darrer */
71        how_many = calc_blocks_with_offset(offset, n_bytes, \
72                                  &block, &f_offset, &l_size);
73        if (how_many < 1) {
74              puts("read_file: error al calcul de la posicio inicial");
75              return n_bytes_written;
76        }
77
78        /* La idea es igual que a la lectura: fer les tres seccions on:
79              Obtenim el bloc logic.
80              Escrivim sobre ell.
81              El tornam a disco.
82        Calculam el canvi de tamany al final de tot. */
83        error = translate_inode(inode, block, &n_block, 1);
84        if(error == -2) {
85              puts("write_file: no queda espai lliure");
86              return n_bytes_written;
87        }
88        bread(n_block, &tmp_block);
89        if (n_bytes < (BLOCK_SIZE - f_offset)) {
90              bytes_copy = n_bytes;
91        } else {
92              bytes_copy = BLOCK_SIZE - f_offset;
93        }
94        memcpy(tmp_block.valor + f_offset, buffer, bytes_copy);
95        bwrite(n_block, &tmp_block);
96        n_blocks_written = 1;
97        n_bytes_written += bytes_copy;
98        how_many--;
```

```
99
100       /* Importantissim: comencam a contar des de un ja que
101         que hem llegit el primer bloc. Aqui treballam amb els blocs
102         intermijos. */
103       for (/*res*/; how_many -1 > 0; how_many--) {
104             block++;
105             error = translate_inode(inode, block, &n_block, 1);
106             if (error == -2) {
107                   puts("write_file: no queda espai lliure");
108                   return n_bytes_written;
109             }
110             bread(n_block, &tmp_block);
111             memcpy(tmp_block.valor,(char*)buffer + n_bytes_written, BLOCK_SIZE);
112             bwrite(n_block, &tmp_block);
113             n_blocks_written++;
114             n_bytes_written += BLOCK_SIZE;
115       }
116
117       /* Comprovam que queda un altre bloc per escriure. */
118       if (how_many == 1) {
119             block++;
120             error = translate_inode(inode, block, &n_block, 1);
121             if(error == -2) {
122                   puts("write_file: no queda espai lliure");
123                   return n_bytes_written;
124             }
125             bread(n_block, &tmp_block);
126             bytes_copy = l_size;
127             memcpy(tmp_block.valor, (char *)buffer + n_bytes_written, bytes_copy);
128             bwrite(n_block, &tmp_block);
129             n_blocks_written++;
130             n_bytes_written += bytes_copy;
131       }
132
133       /* Calcul del tamany de l'inode un cop fetes les modificacions.*/
134       read_inode(inode, &tmp_inode);
135       grow_size = tmp_inode.size - offset - n_bytes_written;
136       if (grow_size < 0) {
137             tmp_inode.size -= grow_size;
138             write_inode(inode, &tmp_inode);
139       }
140       if (n_bytes_written < n_bytes) {
141             puts("write_file: no s'han escrit tots els bytes");
```

```
142         }
143         return n_bytes_written;
144
145 }
146
147 /** Llegeix un cert nombre de bits d'un fitxer.
148  * @inode: nombre d'inode on volem escriure
149  * @buffer: Punter buffer de sortida aquesta funcio reservara l'espai.
150  * @offset: determinam el primer byte del fitxer.
151  * @n_bytes: nombre de bytes a llegir
152  * @return: el nombre de bytes llegits.
153  */
154 int read_file(int inode, void **buf, int offset, int n_bytes) {
155         int j, error;
156         int b_index = 0;
157         int n_block = 0;
158         /* Quants de blocs hem de llegir, nombre del primer bloc, desplacament
159         sobre el primer bloc */
160         int how_many;
161         int block = 0;
162         int f_offset = 0;
163         int l_size = 0;
164         int n_bytes_readen = 0;
165         emofs_block tmp_block;
166
167         how_many = calc_blocks_with_offset(offset, n_bytes, \
168                                 &block, &f_offset, &l_size);
169         n_bytes_readen = 0;
170         if (how_many < 1) {
171                 puts("read_file: error al calcul de la posicio inicial");
172                 return n_bytes_readen;
173         }
174
175
176         error = translate_inode(inode, block, &n_block, 0);
177
178         if (error < 0) {
179                 return 0;
180         }
181         bread(n_block, &tmp_block);
182         if (n_bytes + f_offset < BLOCK_SIZE) {
183                 b_index = n_bytes;
184                 n_bytes_readen = n_bytes;
```

```
185         } else {
186               b_index = BLOCK_SIZE - f_offset;
187               n_bytes_readen = b_index;
188         }
189         /* l'unic coment en que n_bytes_readen no es major a b_index */
190         *buf = malloc(n_bytes_readen*sizeof(char));
191         memcpy(*buf, (tmp_block.valor + f_offset), n_bytes_readen);
192         how_many--;
193
194         if (how_many == 0) {
195               /* Sols havia un bloc per llegir. */
196               return n_bytes_readen;
197         }
198
199         /* Recorda que la variable ««block conte el nombre del primer bloc de
200           dades. Anam a llegir blocs sencers, per aixo descartam el darrer.*/
201         for (/*res*/; (how_many -1) > 0 ; how_many--) {
202               block++;
203               translate_inode(inode, block, &n_block, 0);
204               bread(n_block, &tmp_block);
205               n_bytes_readen += BLOCK_SIZE;
206               *buf = realloc(*buf, n_bytes_readen*sizeof(char));
207               memcpy((char *) *buf + b_index, tmp_block.valor, BLOCK_SIZE);
208               b_index += BLOCK_SIZE;
209         }
210         /* Ara sols queda el darrer bloc. */
211         block++;
212
213         translate_inode(inode, block, &n_block, 0);
214         bread(n_block, &tmp_block);
215         n_bytes_readen += ((l_size) ? l_size : BLOCK_SIZE); /* Hem llegit el darrer
216         *buf = realloc(*buf, n_bytes_readen*sizeof(char));
217         memcpy((char *)*buf + b_index, tmp_block.valor, l_size);
218
219         if (n_bytes_readen != n_bytes) {
220               puts("read_file: error amb el nombre de bytes llegits");
221         }
222         return n_bytes_readen;
223   }
224
225   void truncate_assist(int block, int level, \
226                   emofs_inode *inode, int *n_truncated_blocks);
227
```

```
228 /** Trunca un fitxer a partir del byte n.
229  * Si n_bytes = 0 alliberam tots els blocs.
230  * @inode: el nombre d'inode.
231  * @n_byte: el byte final del fitxer.
232  * @return: 0 si exit.
233  */
234 int truncate_file(int inode, int n_byte) {
235       int how_many, block, f_offset, l_size;
236       int n_block, n_bytes, n_inode;
237       int i;
238       emofs_inode tmp_inode;
239       int blocks_to_truncate;
240
241       read_inode(inode, &tmp_inode);
242
243       if (tmp_inode.size <= n_byte) {
244             /* Si fitxer es mes petit o del tamany a truncar
245              * no s'ha de fer res. */
246             return 0;
247       }
248
249       blocks_to_truncate = (tmp_inode.size-n_byte)/BLOCK_SIZE;
250       for(i = INDIRECT_POINTER_COUNT-1; i >= 0; i--) {
251             if(tmp_inode.indirect_pointer[i] != NULL_POINTER) {
252                   truncate_assist(tmp_inode.indirect_pointer[i], i,
    \
253                                   &tmp_inode, &blocks_to_truncate);
254                   if (blocks_to_truncate > 0) {
255                         free_block(tmp_inode.indirect_pointer[i]);
256                         tmp_inode.indirect_pointer[i] = NULL_POINTER;
257                         tmp_inode.block_count--;
258                         blocks_to_truncate--;
259                   }
260             }
261       }
262       for(i = DIRECT_POINTER_COUNT-1; i >= 0; i--) {
263             /* El blocks_to_truncate > 0 no esta a la condicio
264              * del bucle per donar suport als fitxers esparsos. */
265             if (blocks_to_truncate > 0 && \
266                tmp_inode.direct_pointer[i] != NULL_POINTER) {
267                   free_block(tmp_inode.direct_pointer[i]);
268                   tmp_inode.direct_pointer[i] = NULL_POINTER;
269                   tmp_inode.block_count--;
```

```
270                     blocks_to_truncate--;
271               }
272         }
273
274         tmp_inode.size = n_byte;
275         write_inode(inode, &tmp_inode);
276         return 0;
277   }
278
279   void truncate_assist(int block, int level,
280                   emofs_inode *inode, int *n_truncated_blocks) {
281         int i;
282         int tmp[INDIRECT_POINTERS_PER_BLOCK];
283         if (level <= 0) {
284               free_block(block);
285               (*n_truncated_blocks)--;
286         } else {
287               bread(block, tmp);
288               for(i = 0; i < INDIRECT_POINTERS_PER_BLOCK; i++) {
289                     if(tmp[i] != NULL_POINTER) {
290                           truncate_assist(tmp[i], level-1, \
291                                       inode, n_truncated_blocks);
292                           tmp[i] = NULL_POINTER;
293                           inode->block_count--;
294                     }
295               }
296               bwrite(block, tmp);
297         }
298   }
299
300   /** Obte la informacio d'un inode.
301    * @inode: nombre del fitxer.
302    * @stat: el punter de sortida de les dades.
303    * @return: 0 si exit.
304    */
305   int stat_file(int inode, emofs_inode_stat *stat) {
306         emofs_inode tmp;
307         int error;
308
309         error = read_inode(inode, &tmp);
310         if (error > -1) {
311               memcpy(stat, &tmp, sizeof(emofs_inode_stat));
312         }
```

```
313         return error;
314 }
315
316 /** Canvi el la data de modificacio de l'inode.
317  * @inode: nombre del fitxer
318  * @date: data epoch a posar
319  * @return: 0 si exit
320  */
321 int timestamp_file(int inode, time_t date) {
322         emofs_inode i;
323         read_inode(inode, &i);
324         write_inode_time(inode, &i, date);
325         return 0;
326 }
```

A.2.7 dir.h

```
1  /* Modul dels directoris. Els directoris enmagatzemen la informacio relativa a
2   * l'ordenacio i acces dels fitxers. Son la capa mes externa del sistema de
3   * fitxers i per aixo tenen la nomenclatura «»emofs_x. */
4
5  #include <time.h>
6
7  #include "common.h"
8  #include "file.h"
9
10 typedef struct {
11         char name[MAX_FILENAME_LEN];
12         __u32 inode; /* NULL_POINTER si es una entrada buida */
13 } emofs_dir_entry;
14
15 /** D'una cadena representant una ruta de fitxer obte el cami entre
16  * directoris.
17  * @path: la ruta del fitxer.
18  * @first: cadena que conte la primera part de la ruta.
19  * @last: cadena que conte la resta de la ruta.
20  * @return: 0 si troba
21  * -1 si s'obte un error.
22  */
23 int emofs_extract_path(const char *path, char *head, char *tail);
24
25 /** Torna el directori on s'ha de crear el fitxer/carpeta i el nom
```

```
26  * donant el path sencer: Pe: /dir1/dir2/fx torna: /dir1/dir2 i fx.
27  * @path: path sencer a tallar
28  * @partial_path: torna la primera part del path d'alla on penjara el
29  * fitxer/directori
30  * @new_dir_file: Nom del fitxer/directori a crear
31  * @return: 0 si es un fitxer
32  * 1 si es directori
33  * -1 en cas d'error
34  */
35 int emofs_get_partial_path(const char *path, \
36                     char *partial_path, char *new_dir_file);
37
38 /** Des de l'inode del directori i el cami parcial obte l'inode del directori i
39  * de l'entrada.
40  * Recorda que a la primera crida recursiva sempre li haurem d'indicar
41  * p_inode_dir l'inode del directori arrel.
42  * Conve usar extract_path per obtenir els subcamins.
43  * @path: ruta, parcial o completa, del fitxer que buscam.
44  * @p_inode_dir: l'inoide del directori.
45  * @p_inode: l'inode del fitxer.
46  * @p_entry: entrada del fitxer dins el directori.
47  * @return: 0 si exit.
48 */
49 int emofs_find_entry(const char *path, int *p_inode_dir,
50                 int *p_inode, int *p_entry);
51
52 /** Comprova si un fitxer existeix.
53  * @path: cami a mirar
54  * @return 0 si existeix, -1 si no
55  */
56 int emofs_file_exists(const char *path);
57
58 /** Crea un fitxer o directori i la seva entrada de directori. Te control de
59 * concurrencia. Princpipalment empra les funcions find_entry, alloc_inode,
60 * write_file.
61 * @path: ruta del fitxer a crear
62 * @return: 0 si correctament.
63         -1 en cas d'error.
64 */
65 int emofs_create(const char *path);
66
67 /** Crea l'enllac de una entrada de directori del src_path a l'inode espeificic
68  * per la altre entrada de directori link_path. Actualitza la quatitat
```

```
69  * d'enllacos d'entrades en el directori de l'inode. Te control de
70  * concurrencia.
71  * @src_path: ruta del fitxer a enllacar
72  * @link_path: ruta de l'enllac
73  * @return: 0 si correctament.
74  * -1 en cas d'error.
75  */
76 int emofs_link(const char *src_path, const char *link_path);
77
78 /** Borra l'entrada de directori especificada y en cas de que sigui l'ultim. Te
79  * control de concurrencia.
80  * enllac existent borra el propi fitxer/directori.
81  * @path: ruta del fitxer/directori
82  * @return: 0 si correctament
83  * -1 en cas d'error.
84  */
85 int emofs_unlink(const char *path);
86
87 /** Determina si el cami es d'un directori o fitxer.
88  * @path: ruta del fitxer
89  * @return: 1 si es fitxer
90  * 0 si es directori
91  * -1 si no existex
92  */
93 int emofs_is_file(const char *path);
94
95 /** Canvi el la data de modificacio de l'inode.
96  * @path: cami del fitxer
97  * @date: data epoch a posar
98  * @return: 0 si exit
99  */
100 int emofs_update_time(const char *path, time_t date);
101
102 /** Posa el contingut del directori en un buffer de memoria.
103  * el nom de cada entrava ve separat per ':'.
104  * Es ell qui reserva la memoria de buf i la funcio que el
105  * cridi qui l'ha d'alliberar en haver-ho emprat.
106  * @path: Cami al directori
107  * @buf: llistat de les entrades de directori separades per ':'
108  * @return: numero de d'entrades llistades.
109 */
110 int emofs_dir(const char *path, char **buf);
111
```

```
112 /** De la sortida de emofs_dir (contingut de directori separat per :)
113  * lleva una entrada, i la copia dins filename.
114  * @dir_content: llista d'entrades separades per : . En borra la primera
115  * @filename: hi deixa el nom de l'entrada de directori
116  * @return 0 si lleva entrada (exit)
117  * -1 en cas d'error
118  */
119 int extract_dir_entry(char *dir_content, char *filename);
120
121 /** Obte la informacio d'un fitxer.
122  * @path: Cami al fitxer.
123  * @stat: el punter de sortida de les dades.
124  * @return: 0 si exit.
125 */
126 int emofs_stat(const char *path, emofs_inode_stat *stat);
127
128 /** Llegeix un cert nombre de bits d'un fitxer.
129  * @path: Cami al desti
130  * @buf: punter al buffer de sortida, la funcio s'encarrega de reservar l'espai.
131  * @offset: determinam el primer byte del fitxer.
132  * @n_bytes: nombre de bytes a llegir
133  * @return: el nombre de bytes llegits.
134 */
135 int emofs_read(const char *path, void **buf, int offset, int n_bytes);
136
137 /** Escriu un cert nombre de bits d'un fitxer. Te control de concurrencia.
138  * @path: Cami al desti
139  * @buf: buffer de sortida.
140  * @offset: determinam el primer byte del fitxer.
141  * @n_bytes: nombre de bytes a escriure
142  * @return: el nombre de bytes escriure.
143 */
144 int emofs_write(const char *path, const void *buf, int offset, int n_bytes);
```

A.2.8 dir.c

```
1 #include <string.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #include "dir.h"
6 #include "inode.h"
```

```
7  #include "super.h"
8  #include "sem.h"
9  #include "common.h"
10
11 int mutex = 0;
12
13 /** D'una cadena representant una ruta de fitxer obte el cami entre
14  * directoris.
15  * @path: la ruta del fitxer.
16  * @first: cadena que conte la primera part de la ruta.
17  * @last: cadena que conte la resta de la ruta.
18  * @return: 0 si troba.
19  * -1 si s'obte un error.
20  */
21 int emofs_extract_path(const char *path, char *head, char *tail) {
22 /* Exemples de l'enunciat
23    * camino = "/dir1/dir2/fichero"
24      inicial = "dir1" (devuelve DIRECTORIO)
25      final = "/dir2/fichero"
26
27    * camino = "/dir/"
28      inicial = "dir" (devuelve DIRECTORIO)
29      final = "/"
30
31    * camino = "/fichero"
32      inicial = "fichero" (devuelve FICHERO)
33      final = ""
34  */
35       int delim_found = 0;
36       int i, j, k;
37
38       if (path[0] != '/') {
39             puts("Error extract_path: No hi ha barra al principi");
40             return -1;
41       }
42
43       /* S'ha dignorar la primera barra per aixo i = 1 */
44       j = 0;
45       k = 0;
46       for (i = 1; path[i] != '\0'; i++) {
47             if ((path[i] == '/') && !delim_found) {
48                   delim_found = 1;
49             }
```

```
50              if (!delim_found) {
51                    head[j] = path[i];
52                    j++;
53              } else {
54                    tail[k] = path[i];
55                    k++;
56              }
57        }
58        /* Posam els finals d'string */
59        head[j] = '\0';
60        tail[k] = '\0';
61        return delim_found;
62  }
63
64  /** Torna el directori on s'ha de crear el fitxer/carpeta i el nom
65   * donant el path sencer: Pe: /dir1/dir2/fx torna: /dir1/dir2 i fx.
66   * @path: path sencer a tallar
67   * @partial_path: torna la primera part del path d'alla on penjara el
68   * fitxer/directori
69   * @new_dir_file: Nom del fitxer/directori a crear
70   * @return: 2 si es un fitxer
71   * 1 si es directori
72   * -1 en cas d'error
73   */
74  int emofs_get_partial_path(const char *path,
75                      char *partial_path, char *new_dir_file) {
76
77        int i, j, k;
78        int to_return = -1;
79
80        i = strlen(path)-2;
81        while (path[i] != '/') {
82              i--;
83        }
84        k = 0;
85        if (path[strlen(path)-1] == '/'){
86              to_return = DIRECTORY_INODE; /* Es una directori */
87              for (j = i+1; j < strlen(path)-1; j++){
88                    new_dir_file[k] = path[j];
89                    k++;
90              }
91        } else {
92              to_return = FILE_INODE; /* Es un fitxer */
```

```
93              for (j = i+1; j < strlen(path); j++){
94                    new_dir_file[k] = path[j];
95                    k++;
96              }
97        }
98        new_dir_file[k] = '\0';
99
100       for (j = 0; j < i+1; j++){
101             partial_path[j] = path[j];
102       }
103       partial_path[j] = '\0';
104       return to_return;
105 }
106
107
108 /** Des de l'inode del directori i el cami parcial obte l'inode del directori
109  * i de l'entrada. Recorda que a la primera crida recursiva sempre li haurem
110  * d'indicar p_inode_dir l'inode del directori arrel.
111  * Conve usar extract_path per obtenir els subcamins.
112  * @path: ruta, parcial o completa, del fitxer que buscam.
113  * @p_inode_dir: l'inode del directori.
114  * @p_inode: l'inode del fitxer.
115  * @p_entry: entrada del fitxer dins el directori.
116  * @return: 0 si exit.
117 */
118 int emofs_find_entry(const char *path, int *p_inode_dir,
119                 int *p_inode, int *p_entry) {
120
121       int found;
122       int is_dir;
123       char target [MAX_FILENAME_LEN];
124       char tail [MAX_PATH_LEN];
125       emofs_inode inode;
126       emofs_dir_entry *dir_entry;
127       int dir_entry_count;
128       int i;
129       emofs_superblock sb;
130
131       /* Cas trivial del directori root */
132       if(!strcmp(path, "/")) {
133             sbread(&sb);
134             *p_inode_dir = sb.root_inode;
135             *p_inode = sb.root_inode;
```

```
136             *p_entry = 0;
137             return 0;
138         }
139
140         emofs_extract_path(path, target, tail);
141
142         read_inode(*p_inode_dir, &inode);
143         /* Cal recordar que no tenim desada enlloc la quantitat d'entrades que
144          * tenim dins el directori. Amb la divisio tamany_escrit/tamany_dades
145          * obtindrem quants d'elements tenim. */
146         dir_entry_count = inode.size / sizeof(emofs_dir_entry);
147         /* p_inode_dir -> inode del pare
148          * dir_entry -> l'entrada de directori: nom+inode
149          * dir_entry_count -> nombre de l'entrada dins el directori pare
150          */
151         found = 0;
152         for (i = 0; i < dir_entry_count; i++) {
153             read_file(*p_inode_dir, (void *)&dir_entry, i*sizeof(emofs_dir_entry),
154                     sizeof(emofs_dir_entry));
155             found = !strcmp(target, dir_entry->name);
156             if (found) {
157                 break;
158             }
159         }
160         if (!found) {
161             return -1;
162         }
163
164         *p_entry = i;
165         *p_inode = dir_entry->inode;
166         /* Si encara queda cami seguim amb les crides recursives */
167         if (strcmp(tail, "")) {
168             *p_inode_dir = dir_entry->inode;
169             free(dir_entry);
170             return emofs_find_entry(tail, p_inode_dir, p_inode, p_entry);
171         }
172
173         free(dir_entry);
174
175         /* Si arribam aqui s'aturen les crides recursives. */
176         return 0;
177 }
178
```

```
179 /** Crea un fitxer o directori i la seva entrada de directori.
180 * Princpipalment empra les funcoons find_entry, alloc_inode, write_file
181 * @path: ruta del fitxer a crear
182 * @return: 0 si es crea correctament.
183 * -1 en cas d'error.
184 */
185 int emofs_create(const char *path) {
186       emofs_superblock sb;
187       int i = 0;
188       int p_inode_dir;
189       int p_inode;
190       int p_new_inode;
191       int p_entry;
192       emofs_inode inode;
193       int type;
194       char partial_path[MAX_PATH_LEN];
195       char new_dir_file[MAX_FILENAME_LEN];
196       int error = 0;
197       int dir_entry_count;
198       emofs_dir_entry dir_entry;
199
200       if(!mutex){
201             emofs_sem_get(&mutex);
202       }
203       emofs_sem_wait(mutex);
204
205       sbread(&sb);
206       if (emofs_file_exists(path)) {
207             /* Existeix, per tant no el podem crear */
208             puts("emofs_create: El fitxer o directori ja existeix");
209             emofs_sem_signal(mutex);
210             return -1;
211       }
212
213       type = emofs_get_partial_path(path, partial_path, new_dir_file);
214
215       /* Comprovam que existeixi la ruta on crear el fitxer o directori */
216       p_inode_dir = sb.root_inode;
217       if (emofs_find_entry(partial_path, \
218                       &p_inode_dir, &p_inode, &p_entry) == -1) {
219             puts("emofs_create: El directori on es preten fer " \
220                 "la creacio no existeix");
221             printf("partial_path %s\n", partial_path);
```

```
222             printf("p_inode_dir: %d, p_inode: %d, p_entry: %d\n", \
223                  p_inode_dir, p_inode, p_entry);
224             emofs_sem_signal(mutex);
225             return error;
226       }
227
228       /* Obtenim el directori on volem crear l'entrada */
229       p_inode_dir = sb.root_inode;
230       emofs_find_entry(path, &p_inode_dir, &p_inode, &p_entry);
231
232       dir_entry.inode = alloc_inode(type); /* Aqui reservam l'inode */
233       if (dir_entry.inode == -1) {
234             emofs_sem_signal(mutex);
235             return -1;
236       }
237       for(i = 0; i < MAX_FILENAME_LEN; i++) {
238             dir_entry.name[i] = ' ';
239       }
240       strcpy(dir_entry.name, new_dir_file);
241
242       /* Llegim inode del directori del pare per saber el seu tamany i poder
243        * escriure al final. */
244       read_inode(p_inode_dir, &inode);
245       error = write_file(p_inode_dir, &dir_entry, inode.size, \
246                     sizeof(emofs_dir_entry));
247       if ( error < 0) {
248             puts("emofs_create: Error d'escriptura");
249             emofs_sem_signal(mutex);
250             return error;
251       }
252       emofs_sem_signal(mutex);
253       return error;
254 }
255
256 /** Comprova si un fitxer existeix.
257  * @path: cami a mirar
258  * @return 0 si existeix, -1 si no
259  */
260 int emofs_file_exists(const char *path) {
261       int zero = 0;
262       int tmp1, tmp2;
263       return !emofs_find_entry(path, &zero, &tmp1, &tmp2);
264 }
```

```
265
266 /** Crea l'enllac de una entrada de directori del src_path a l'inode espeificic
267  * per la altre entrada de directori link_path. Actualitza la quatitat
268  * d'enllacos d'entrades en el directori de l'inode. Te control de
269  * concurrencia.
270  * @src_path: ruta del fitxer a enllacar
271  * @link_path: ruta de l'enllac
272  * @return: 0 si correctament.
273  * -1 en cas d'error.
274  */
275 int emofs_link(const char *src_path, const char *link_path) {
276       int p_dir_src_inode = 0;
277       int p_src_inode = 0;
278       int p_dir_dst_inode = 0;
279       int p_dst_inode = 0;
280       int p_entry = 0;
281
282       emofs_inode inode;
283       emofs_dir_entry dir_entry;
284
285       char link_name[MAX_FILENAME_LEN];
286       char partial_link_path[MAX_PATH_LEN];
287
288       if (!mutex){
289             emofs_sem_get(&mutex);
290       }
291       emofs_sem_wait(mutex);
292
293       emofs_get_partial_path(link_path, partial_link_path, link_name);
294       /* Comprovam i obtenim informacio del primer path */
295       p_dir_src_inode = 0;
296       if (emofs_find_entry(src_path, \
297                   &p_dir_src_inode, &p_src_inode, &p_entry) == -1) {
298             puts("EmoFS_Link: El fitxer o directori font no existeix");
299             emofs_sem_signal(mutex);
300             return -1;
301       }
302
303       /* Comprovam que no estiqui ja creat el fitxer */
304       p_dir_dst_inode = 0;
305       if (emofs_find_entry(link_path, \
306                   &p_dir_dst_inode, &p_dst_inode, &p_entry) == 0) {
307             puts("EmoFS_Link: El fitxer ja existeix");
```

```
308             emofs_sem_signal(mutex);
309             return -1;
310       }
311
312       /* Incrementam el numero d'enllacos de l'inode font */
313       read_inode(p_src_inode, &inode);
314       inode.link_count++;
315       write_inode(p_src_inode, &inode);
316
317       /* Cream l'entrada de directori de l'enllac */
318       dir_entry.inode = p_src_inode;
319       strcpy(dir_entry.name, link_name);
320
321       /* Anam a escriure l'entrada de directori al direcotri desti */
322       read_inode(p_dir_dst_inode, &inode); /* Per saber el tamany de fitxer */
323       if (write_file(p_dir_dst_inode, &dir_entry, \
324                   inode.size, sizeof(emofs_dir_entry)) == -1) {
325             puts("EmoFS_Link: Error d'escriptura");
326             emofs_sem_signal(mutex);
327             return -1;
328       }
329       emofs_sem_signal(mutex);
330       return 0;
331 }
332
333 /** Borra l'entrada de directori especificada y en cas de que sigui l'ultim
334  * enllac existent borra el propi fitxer/directori. Te control de concurrencia.
335  * @path: ruta del fitxer/directori
336  * @return: 0 si correctament
337  * -1 en cas d'error.
338  */
339 int emofs_unlink(const char *path) {
340       int p_inode = 0;
341       int p_dir_inode = 0;
342       int p_entry = 0;
343
344       emofs_inode inode;
345       emofs_dir_entry *last_dir_entry;
346
347       if(!mutex){
348             emofs_sem_get(&mutex);
349       }
350       emofs_sem_wait(mutex);
```

```
351         if (emofs_find_entry(path, &p_dir_inode, &p_inode, &p_entry) == -1) {
352               puts("EmoFS_UnLink: El fitxer o directori font no existeix");
353               emofs_sem_signal(mutex);
354               return -1;
355         }
356
357         read_inode(p_dir_inode, &inode);
358         /* Borram entrada de directori */
359         if (inode.size > sizeof(emofs_dir_entry)) {
360               /* Hi ha mes entrades posam l'ultima al lloc de la que volem
361                * borrar. */
362               read_file(p_dir_inode, (void *)&last_dir_entry, \
363                       inode.size-sizeof(emofs_dir_entry), \
364                       sizeof(emofs_dir_entry));
365               write_file(p_dir_inode, last_dir_entry, \
366                        p_entry*sizeof(emofs_dir_entry), \
367                        sizeof(emofs_dir_entry));
368         }
369
370         free(last_dir_entry);
371
372         /* Truncam per l'ultima entrada, si sols hi ha la que volem borrar be
373          * sino llevam l'ultima que ara estara copiada al lloc de la que voliem
374          * borrar. */
375         truncate_file(p_dir_inode, inode.size-sizeof(emofs_dir_entry));
376         read_inode(p_inode, &inode);
377         if (inode.link_count == 0) {
378               /* Hem de alliberar blocs de dades i inode */
379               truncate_file(p_inode, 0);
380               free_inode(p_inode);
381         } else {
382               inode.link_count--;
383               write_inode(p_inode, &inode);
384         }
385         emofs_sem_signal(mutex);
386         return 0;
387   }
388
389   /** Determina si el cami es d'un directori o fitxer.
390    * @path: ruta del fitxer
391    * @return: 0 si es directori
392    * 1 si es fitxer
393    */
```

```
394 int emofs_is_file(const char *path) {
395       emofs_inode_stat info;
396
397       emofs_stat(path, &info);
398       return info.type == FILE_INODE;
399
400 }
401
402 /** Canvi el la data de modificacio de l'inode.
403  * @path: cami del fitxer
404  * @date: data epoch a posar
405  * @return: 0 si exit
406  */
407 int emofs_update_time(const char *path, time_t date) {
408       int inode = 0;
409       int inode_dir = 0;
410       int entry = 0;
411       emofs_find_entry(path, &inode_dir, &inode, &entry);
412       timestamp_file(inode, date);
413       return 0;
414 }
415
416 /** Posa el contingut del directori en un buffer de memoria.
417  * el nom de cada entrava ve separat per ':'.
418  * Es ell qui reserva la memoria de buf i l'ha d'alliberar
419  * la funcio que l'empri en haver acabat.
420  * @path: Cami al directori
421  * @buf: llistat de les entrades de directori separades per ':'
422  * @return: numero de d'entrades llistades, -1 error, -2 era un fitxer.
423 */
424 int emofs_dir(const char *path, char **buf) {
425       int p_inode;
426       emofs_inode inode;
427       emofs_superblock sb;
428       int p_entry; /* No s'empra pero es necessari per cridar find_entry */
429       int p_inode_dir;
430       int dir_entry_count;
431       emofs_dir_entry *dir_entry;
432       int error;
433       int i, j, k;
434       int used_entry_count = 0;
435       int buffer_size = 0;
436
```

```
437         sbread(&sb);
438         p_inode_dir = sb.root_inode;
439         error = emofs_find_entry(path, &p_inode_dir, &p_inode, &p_entry);
440
441         if (error < 0) {
442               return error;
443         }
444
445         read_inode(p_inode, &inode);
446         dir_entry_count = inode.size/sizeof(emofs_dir_entry);
447
448         *buf = malloc(sizeof(char));
449
450         k = 0;
451         for (i = 0; i < dir_entry_count; i++) {
452               read_file(p_inode, (void *) &dir_entry, \
453                     i*sizeof(emofs_dir_entry), sizeof(emofs_dir_entry));
454               if (dir_entry->inode != NULL_POINTER) {
455                     buffer_size += strlen(dir_entry->name)+1; /*Mes les ':'*/
456                     /*Memoria dinamica, el que faltava per fer*/
457                     *buf = realloc(*buf, buffer_size*sizeof(char));
458                     if (*buf == NULL) {
459                           puts("Error redimensionant tamany de buffer.");
460                     }
461                     for(j = 0; dir_entry->name[j] != '\0'; j++) {
462                           (*buf)[k] = dir_entry->name[j];
463                           k++;
464                     }
465                     (*buf)[k] = ':'; /* separadors de nom de fitxer */
466                     k++;
467                     used_entry_count++;
468               }
469         }
470
471         free(dir_entry);
472
473         if (used_entry_count > 0) {
474               /* Posam el final d'string substituint el separador */
475               (*buf)[k-1] = '\0';
476         } else {
477               *buf = malloc(sizeof(char));
478               (*buf)[k] = '\0'; /* k sera 0 */
479         }
```

```
480
481         return used_entry_count;
482 }
483
484 /** De la sortida de emofs_dir (contingut de directori separat per ':')
485  * lleva una entrada i la copia dins filename.
486  * @dir_content: llista d'entrades separades per ':'; En borra la primera
487  * @filename: hi deixa el nom de l'entrada de directori
488  * @return 0 si lleva entrada (exit)
489  * -1 en cas d'error
490  */
491 int emofs_extract_dir_entry(char *dir_content, char *filename) {
492         int i = 0;
493         int j = 0;
494         char tmp;
495
496         /* Copiam la darrera entrada. El truc ve de que desarem el contingut de
497          * dir_content dins filename llegint al reves i despres el
498          * girarem. */
499         /* abcde:fgh:ijk -> filename = kji */
500         i = strlen(dir_content)-1;
501         j = 0;
502         for(; dir_content[i] != ':' && i >= 0; i--) {
503                 if (j >= MAX_FILENAME_LEN) {
504                         puts("emofs_extract_dir_entry: nom massa llarg");
505                         return -1;
506                 }
507                 filename[j] = dir_content[i];
508                 dir_content[i] = '\0';
509                 j++;
510         }
511         /* Com que la condicio es que haguem trobat un «»: sabem que a
512          * dir_content en queda com a minim un. El llevam de la sortida.
513          */
514         if (i > 0) {
515                 dir_content[i] = '\0';
516         }
517         /* Posam el nom en l'ordre correcte doncs tenim filename a
518          * l'inreves. Posam tambe un caracter null al final. */
519         j--;
520         for (i = 0; i < (j/2)+1; i++) {
521                 tmp = filename[i];
522                 filename[i] = filename[j-i];
```

```
523             filename[j-i] = tmp;
524       }
525       j++;
526       filename[j] = '\0';
527       return 0;
528 }
529
530
531 /** Obte la informacio d'un fitxer o directori
532  * @path: Cami al fitxer/directori.
533  * @stat: el punter de sortida de les dades.
534  * @return: 0 si exit.
535 */
536 int emofs_stat(const char *path, emofs_inode_stat *stat) {
537       int inode = 0;
538       emofs_superblock sb;
539       int p_entry = 0; /* No s'empra pero es necessari per cridar find_entry */
540       int p_inode_dir = 0;
541       int error = 0;
542
543       sbread(&sb);
544       p_inode_dir = sb.root_inode;
545       error = emofs_find_entry(path, &p_inode_dir, &inode, &p_entry);
546       if(error < 0) {
547             puts("emofs_stat: no s'ha trobat entrada");
548             return error;
549       }
550       if(inode < 0) {
551             puts("emofs_stat: inode negatiu");
552             return -1;
553       }
554       error = stat_file(inode, stat);
555       return error;
556 }
557
558 /** Llegeix un cert nombre de bits d'un fitxer.
559  * @path: Cami al desti
560  * @buf: punter al buffer de sortida, la funcio s'encarrega de reservar l'espai.
561  * @offset: determinam el primer byte del fitxer.
562  * @n_bytes: nombre de bytes a llegir
563  * @return: el nombre de bytes llegits.
564 */
565 int emofs_read(const char *path, void **buf, int offset, int n_bytes) {
```

```
566         int inode;
567         emofs_superblock sb;
568         int p_entry; /* No s'empra pero es necessari per cridar find_entry */
569         int p_inode_dir;
570 
571         sbread(&sb);
572         p_inode_dir = sb.root_inode;
573         emofs_find_entry(path, &p_inode_dir, &inode, &p_entry);
574         return read_file(inode, buf, offset, n_bytes);
575 }
576 
577 /** Escriu un cert nombre de bits d'un fitxer. Te control de concurrencia.
578  * @path: Cami al desti
579  * @buf: buffer de entrada.
580  * @offset: determinam el primer byte del fitxer.
581  * @n_bytes: nombre de bytes a escriure
582  * @return: el nombre de bytes escriure.
583 */
584 int emofs_write(const char *path, const void *buf, int offset, int n_bytes) {
585         int inode, error;
586         emofs_superblock sb;
587         /* No s'empra pero es necessari per cridar find_entry */
588         int p_entry;
589         int p_inode_dir;
590 
591         if(!mutex){
592               emofs_sem_get(&mutex);
593         }
594         emofs_sem_wait(mutex);
595         sbread(&sb);
596         p_inode_dir = sb.root_inode;
597         emofs_find_entry(path, &p_inode_dir, &inode, &p_entry);
598         error = write_file(inode, buf, offset, n_bytes);
599         emofs_sem_signal(mutex);
600         return error;
601 }
```

A.3 Eines de sistema

A.3.1 sem.h

```
1 /* Modul de semafors per control de concurrencia. El funcionament tipic
```

```
 * consisteix en que la funcio obtingui el punter al semafor amb sem_get,
 * demani acces amb un sem_wait i avisi que ha acabat amb un sem_signal. Quan
 * s'executa una simulacio o hi ha la possibilitat de multiples clients cal
 * inicialitzar els semafor amb sem_init. En el cas de les eines d'usuari no es
 * necessari doncs son mono-fil i es suposa que no feim mes d'una tasca a la
 * vegada. */

/** Crea un semafor.
 * @mutex: punter al semafor
 */
void emofs_sem_init(int *mutex);

/** Destrueix un semafor.
 * @mutex: punter al semafor
 */
void emofs_sem_del(int mutex);

/** Obte el punter d'un semafor.
 * @mutex: punter al semafor
 */
void emofs_sem_get(int *mutex);

/** Feim una peticio d'acces a la seccio critica
 * @mutex: punter al semafor
 */
void emofs_sem_wait(int mutex);

/** Assenyalam que hem acabat les tasques de la seccio critica
 * @mutex: punter al semafor
 */
void emofs_sem_signal(int mutex);
```

A.3.2 sem.c

```

#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>

#include "sem.h"
```

```
9
10 #define SEM_KEY_PATH "/"
11
12 /* Obte la clau comuna pel semafor */
13 int get_key(void) {
14       return ftok(SEM_KEY_PATH, 1);
15 }
16
17 /** Crea un semafor.
18  * @mutex: punter al semafor
19  */
20 void emofs_sem_init(int *mutex){
21       emofs_sem_get(mutex);
22       if(*mutex < 0) {
23             *mutex = semget(get_key(), 1, 0600 | IPC_CREAT);
24             semctl(*mutex,0,SETVAL,1);
25       }
26 }
27
28 /** Destrueix un semafor.
29  * @mutex: punter al semafor
30  */
31 void emofs_sem_del(int mutex) {
32       semctl(mutex, 0, IPC_RMID, 0);
33 }
34
35 /** Obte el punter d'un semafor.
36  * @mutex: punter al semafor
37  */
38 void emofs_sem_get(int *mutex){
39       *mutex = semget(get_key(), 1, 0600);
40 }
41
42 /** Feim una peticio d'acces a la seccio critica
43  * @mutex: punter al semafor
44  */
45 void emofs_sem_wait(int mutex){
46       struct sembuf op_P [] = {0, -1, 0};
47       semop(mutex, op_P, 1);
48 }
49
50 /** Assenyalam que hem acabat les tasques de la seccio critica
51  * @mutex: punter al semafor
```

```
52  */
53 void emofs_sem_signal(int mutex){
54       struct sembuf op_V [] = {0, 1, 0};
55       semop(mutex, op_V, 1);
56 }
```

A.4 Simulador

A.4.1 sim.c

```
1  /** Simulador de la practica del sistema de fitxers. El programa consisteix en
2   * crear una imatge de sf, crear un seguit de fils i anar escrivint adins el
3   * sistema.
4   */
5
6  #include <signal.h>
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <sys/ipc.h>
11 #include <sys/sem.h>
12 #include <sys/types.h>
13 #include <sys/wait.h>
14 #include <time.h>
15 #include <unistd.h>
16
17 #include "dir.h"
18 #include "block.h"
19
20 /* Quants de fils escriptors llancarem */
21 #define PROCESS_NUMBER 100
22 /* Nombre d'escriptures que fara un fil escriptor */
23 #define CHILDREN_WRITTING 5
24
25 /* El temps a esperar entre fill i fill, en nanosegons */
26 #define NEXT_CHILDREN_WAIT 1000000
27 /* El temps d'espera d'un fill a fer la üsegent escriptura */
28 #define CHILDREN_WAIT_TIME 100000
29 #define MAX 2000
30
31 /* Necessitarem contar el nombre de fills que ja han acabat */
32 static int ENDED_CHILDREN = 0;
```

```
33 static char SIM_PATH[256];
34 static int mutex = 0;
35
36 /* Classic foser o enterrador. Compta el nombre de fills morts per saber quan
37  * podem aturar el programa.
38  */
39 void reaper() {
40       while(wait3(NULL, WNOHANG, NULL) > 0) {
41             ENDED_CHILDREN++;
42       }
43 }
44
45 /* Crea el directori dins del sistema de fitxers segons el moment actual. Es de
46  * la forma "simul_aaaammddhhmmss"
47  */
48 int init_fs() {
49       time_t t_time;
50       struct tm *t;
51       time(&t_time);
52       t = localtime(&t_time);
53       sprintf(SIM_PATH, "/simul_%d%d%d%d%d%d/",
54             1900 + t->tm_year, 1 + t->tm_mon, t->tm_mday,
55             t->tm_hour, t->tm_min, t->tm_sec);
56       emofs_create(SIM_PATH);
57
58       /**/
59       /* puts("ruta a crear"); */
60       /* puts(SIM_PATH); */
61       /**/
62
63       return 0;
64 }
65
66 /* Feina dels fils escriptors. Primer creen un directori de la forma
67 "proceso_n", on n es el seu PID. Alla fara un prueba.dat. Cada 0,1s han de fer
68 una escriptura "hh:mm:ss Escritura numero i a partir de la posicion j". i es el
69 nombre d'escriptura i j la posicio, la qual es aleatoria. */
70 int sim_work() {
71       int i = 0;
72       char local_path[256];
73       char buff[256];
74       time_t t_time;
75       struct tm *t;
```

```
76        int offset;
77        printf("Inici worker %d\n", getpid());
78
79        strcpy(local_path, SIM_PATH);
80        sprintf(buff, "process_%d/", getpid());
81        strcat(local_path, buff);
82        emofs_create(local_path);
83        /* Un cop creat el directori podem determinar el nom final */
84        strcat(local_path, "prueba.dat");
85        emofs_create(local_path);
86        /**/
87        /* puts("sim: worker: fitxer local"); */
88        /* puts(local_path); */
89        /**/
90        srand(time(NULL));
91        for(i = 0; i < CHILDREN_WRITTING; i++){
92              offset = rand() % MAX;
93              time(&t_time);
94              t = localtime(&t_time);
95              sprintf(buff, "%d:%d:%d escriptura nombre %d a la posicio %d\n",
96                    t->tm_hour, t->tm_min, t->tm_sec, i, offset);
97              emofs_write(local_path, buff, offset, strlen(buff));
98        }
99        /**/
100       printf("sim: worker: final client amb pid %d\n", getpid());
101       /**/
102       return 0;
103 }
104
105
106 void mi_ls(char *path) {
107       int i, j;
108       char msg[MAX_PATH_LEN];
109       char full_path[MAX_PATH_LEN];
110       char partial_path[MAX_PATH_LEN];
111       char filename[MAX_FILENAME_LEN];
112       emofs_inode_stat stat;
113       int entries_count;
114       char type [2];
115       int len;
116       char *dir_content; /* Reserva la memoria la funcio que el torna */
117       /* El nombre de parametres es tots els arguments del programa manco el
118        * propi programa. */
```

```
119
120       sprintf(msg, "Type \t Size \t\t Epoc \t\t Name \n");
121       fwrite(msg, strlen(msg), 1, stdout);
122
123       if (emofs_is_file(path)) {
124             /* ls d'un fitxer mostra sols el path del fitxer */
125             emofs_stat(path, &stat);
126             emofs_get_partial_path(path, partial_path, filename);
127             sprintf(msg, "f \t %d \t\t %d \t %s \n",
128                   stat.size, stat.mtime, filename);
129             fwrite(msg, strlen(msg), 1, stdout);
130       } else {
131             entries_count = emofs_dir(path, &dir_content);
132             for (j=0; j < entries_count; j++) {
133                   emofs_extract_dir_entry(dir_content, filename);
134                   /*Afegim path a l'element per poder fer el
135                    * emofs_stat */
136                   if (path[strlen(path)-1] != '/') {
137                         sprintf(full_path, "%s/%s", path, \
138                               filename);
139                   } else {
140                         sprintf(full_path, "%s%s", path, \
141                               filename);
142                   }
143                   emofs_stat(full_path, &stat);
144                   sprintf(msg, "%c \t %d \t\t %d \t %s \n", \
145                         ((stat.type==FILE_INODE) ? 'f' : 'd'), \
146                         stat.size, stat.mtime, filename);
147                   len = strlen(msg);
148                   fwrite(msg, len, 1, stdout);
149             }
150       }
151 }
152
153 void mi_cat(char *path) {
154       emofs_inode_stat info;
155       int i;
156       char *buffer;
157       emofs_stat(path, &info);
158       for(i = 0; i < info.size; i++) {
159             emofs_read(path, (void *)&buffer, i, 1);
160             fwrite(buffer, 1, 1, stdout);
161       }
```

```
162       free(buffer);
163 }
164
165 /* Imprimeix el llistat de tots els fitxers que s'han generat amb els seus
166  * tamanys. Mostra el contigut d'un fitxer a l'atzar. */
167 int show_work() {
168       int i, j, max;
169       char path[256];
170       char tmp[256];
171       emofs_dir_entry *dir_entry;
172
173       for(j=strlen(SIM_PATH)-1; SIM_PATH[j] == '/' && j > 0; j--) {
174             SIM_PATH[j] = '\0';
175       }
176
177       mi_ls(SIM_PATH);
178
179
180       /* Despres de fer el emofs_dir s'obte una llista de fitxers separats
181        * per dos punts. Basta agafar el primer i imprimir el contigut */
182       emofs_read(SIM_PATH,(void *)&dir_entry, 0, sizeof(emofs_dir_entry));
183       sprintf(path, "%s/%s/prueba.dat", SIM_PATH, dir_entry->name);
184       free(dir_entry);
185
186       /* */
187       puts("");
188       puts("sim: anam a mostrar un fitxer dels creats");
189       puts(path);
190       puts("-----------------------------------");
191       /* */
192       mi_cat(path);
193       /* */
194       puts("-----------------------------------");
195       /* */
196
197       return 0;
198 }
199
200 int main() {
201       int i;
202       /* mi_mount */
203       emofs_sem_init(&mutex);
204       bmount();
```

```
205
206         init_fs();
207         signal(SIGCHLD, reaper);
208         for(i = 0; i < PROCESS_NUMBER; i++) {
209               if (fork() == 0) {
210                     sim_work();
211                     exit(0);
212               } else {
213                     usleep(NEXT_CHILDREN_WAIT);
214               }
215         }
216         while (ENDED_CHILDREN < PROCESS_NUMBER) {
217               pause();
218         }
219         show_work();
220         puts("Simulacio acabada");
221
222         /* mi_umount */
223         bumount();
224
225         emofs_sem_get(&mutex);
226         emofs_sem_del(mutex);
227
228         return 0;
229   }
```

A.5 Eines d'usuari

A.5.1 mi_append.c

```
1  #include <string.h>
2
3  #include "common.h"
4  #include "dir.h"
5
6  /** Escriu al final del fitxer especificat al primer paramentre
7   * la cadena del segon paramentre.
8   * mi_apend <path> <string> */
9  int main(int argc, char **argv) {
10       char path[MAX_PATH_LEN];
11       emofs_inode_stat info;
12
```

```
13        if (argc != 3) {
14              puts("Nombre de parametres incorrecte.\n");
15              puts("Us: mi_append <fitxer> <cadena>\n");
16              return -1;
17        }
18        bmount();
19
20        strcpy(path, argv[1]);
21        if (!emofs_file_exists(path)) {
22              emofs_create(path);
23        }
24
25        if (emofs_is_file(path) == 0) {
26              puts("append: no es poden fer appends a directoris");
27              bumount();
28              return -2;
29        }
30        emofs_stat(path, &info);
31        emofs_write(path, argv[2], info.size, strlen(argv[2]));
32        bumount();
33        return 0;
34  }
```

A.5.2 mi_mkfs.c

```
1   #include <stdlib.h>
2   #include <unistd.h>
3   #include <sys/types.h>
4   #include <sys/stat.h>
5   #include <fcntl.h>
6   #include <string.h>
7
8   #include "block.h"
9   #include "inode.h"
10
11  #include "block_lib.h"
12
13  #include "dir.h"
14
15  #define DEFAULT_BLOCKS 1000
16
17  /* Some fun */
```

```
18 #define AUTHOR_COUNT 2
19 #define AUTHOR_B "bartomeumiro.ascii"
20 #define AUTHOR_P "paurullan.ascii"
21
22 /*
23  * Creacio del fitxer on es te el sistema de fitxers.
24  * mi_mkfs <cantidad_bloques>
25  * codis de retorn:
26  *  0: correcte
27  *  -1: nombre de parametres incorrecte
28  */
29 int main(int argc, char **argv) {
30       int i = 0;
31       int nombre_blocs;
32       int n_inode;
33       emofs_superblock sb;
34       emofs_inode inode;
35       /* Some fun */
36       char author[][20] = { AUTHOR_B, AUTHOR_P };
37       char buf[80];
38       char path[MAX_PATH_LEN];
39       int ascii_art; /* ASCII art, descriptor de fitxer */
40       int pos;
41       int read_bytes;
42
43       emofs_block bloc_zero = block_of_zero();
44
45       if (argc != 2) {
46             nombre_blocs = DEFAULT_BLOCKS;
47       } else {
48             nombre_blocs = atoi(argv[1]);
49       }
50
51       bmount();
52
53       /* Inicialitzam tot el sistema de fitxers */
54       for (i = 0; i < nombre_blocs; i++) {
55             bwrite(i, &bloc_zero);
56       }
57
58       init_superblock(nombre_blocs);
59       init_bitmap();
60       init_inode_array();
```

```
61
62        /* Problema de bootstrap. Es necessita una ruta per crear un fitxer. */
63        n_inode = alloc_inode(DIRECTORY_INODE);
64        sbread(&sb);
65        sb.root_inode = n_inode;
66        sbwrite(&sb);
67
68        /* JUST FOR FUN */
69        /* Cream directori /authors */
70        if (fork() == 0) {
71              execl("mi_mkdir", "mi_mkdir", "/authors", (char *)0);
72        } else {
73              wait3(NULL, 0, NULL);
74        }
75        /* Cream i omplim /authors/author[i].ascii" */
76        for (i = 0; i < AUTHOR_COUNT; i++) {
77              ascii_art = open(author[i], O_RDONLY, S_IRUSR);
78              strcpy(path, "/authors/");
79              strcat(path, author[i]);
80              emofs_create(path);
81              pos = 0;
82              read_bytes = 1; /* per entrar al bucle */
83              while (read_bytes > 0) { /* Sortira amb el break */
84                    lseek(ascii_art, pos, SEEK_SET);
85                    read_bytes = read(ascii_art, buf, 80);
86                    if (!read_bytes) break;
87                    emofs_write(path, buf, pos, read_bytes);
88                    pos += read_bytes;
89              }
90
91
92              close(ascii_art);
93        }
94        /* ending fun */
95
96        bumount();
97
98        return 0;
99  }
```

A.5.3 mi_cat.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "common.h"
#include "dir.h"

#define C_BUF 2048

/*
 * Mostra el contingut d'un fitxer. Aquest cat es especial i permet mostrar el
 * contingut d'un directori.
 * mi_cat <nom>
 * @return: 0 exit, -1 nombre de parametres incorrecte
 */
int main(int argc, char **argv) {
	char path[MAX_PATH_LEN];
	int n_entries;
	int error = 0;
	emofs_inode_stat info;
	char *buffer;
	int offset = 0;
	int bytes_left = 0;
	int to_read = 0; /* bytes llegits */
	int read_bytes = 0;

	if (argc != 2) {
		puts("Nombre de parametres incorrecte.");
		puts("Us: mi_cat <nom>");
		return -1;
	}

	strcpy(path, argv[1]);
	bmount();

	if(!emofs_file_exists(path)) {
		bumount();
		return -2;
	}

	emofs_stat(path, &info);
	for (bytes_left = info.size; bytes_left > 0; bytes_left -= C_BUF) {
		to_read = (bytes_left < C_BUF) ? (bytes_left % C_BUF) : C_BUF;
```

```
44              read_bytes = emofs_read(path, (void *) &buffer, offset, to_read);
45              fwrite(buffer, read_bytes, 1, stdout);
46              /* A la ultima iteracio no es cert pero ens estalvam un acces
47               * a memoria a cada iteracio. */
48              offset += C_BUF;
49      }
50      puts("");
51      bumount();
52      return error;
53  }
```

A.5.4 mi_ln.c

```
1   #include <stdlib.h>
2   #include <string.h>
3
4   #include "common.h"
5
6   /*
7    * Fes un enllac simbolic d'un fitxer.
8    * mi_ln <origen> <desti>
9    * @return: 0 exit, -1 nombre de parametres incorrecte
10   */
11  int main(int argc, char **argv) {
12      char from[MAX_PATH_LEN];
13      char to[MAX_PATH_LEN];
14      int error = 0;
15
16      if (argc != 3) {
17              puts("Nombre de parametres incorrecte.");
18              puts("Us: mi_ln <origen> <desti>");
19              return -1;
20      }
21
22      strcpy(from, argv[1]);
23      strcpy(to, argv[2]);
24      bmount();
25      if (!emofs_file_exists(from)) {
26              puts("El fitxer origen no existeix");
27              error = -2;
28      } else if (emofs_file_exists(to)) {
29              puts("El fitxer desti existeix");
```

```
30              error = -3;
31        } else {
32              emofs_link(from, to);
33        }
34        bumount();
35        return error;
36  }
```

A.5.5 mi_ls.c

```
1   #include <stdlib.h>
2   #include <string.h>
3   #include <stdio.h>
4
5   #include "common.h"
6   #include "dir.h"
7
8   /*
9    * Llista les entrades de directori. (es comporta com un ls -l de unix)
10   * mi_ls <cami>
11   * @return: 0 exit, -1 nombre de parametres incorrecte
12   */
13  int main(int argc, char **argv) {
14        char path[MAX_PATH_LEN];
15        int i, j;
16        char msg[MAX_PATH_LEN];
17        char full_path[MAX_PATH_LEN];
18        char partial_path[MAX_PATH_LEN];
19        char filename[MAX_FILENAME_LEN];
20        emofs_inode_stat stat;
21        int entries_count;
22        char type [2];
23        int len;
24        char *dir_content; /* Reserva la memoria la funcio que el torna */
25        /* El nombre de parametres es tots els arguments del programa manco el
26         * propi programa. */
27        int how_many_params = argc - 1;
28        if (argc < 2) {
29              puts("Nombre de parametres incorrecte.");
30              puts("Us: mi_ls <cami>");
31              return -1;
32        }
```

```
33
34        memset(msg, '\0', MAX_PATH_LEN);
35        memset(path, '\0', MAX_PATH_LEN);
36        memset(full_path, '\0', MAX_PATH_LEN);
37        memset(partial_path, '\0', MAX_PATH_LEN);
38        memset(filename, '\0', MAX_FILENAME_LEN);
39
40        bmount();
41
42        for (i=0; i < how_many_params; i++){
43              strcpy(path, argv[1+i]);
44              for(j=strlen(path)-1; path[j] == '/' && j > 0; j--) {
45                    path[j] = '\0';
46              }
47
48              if(!emofs_file_exists(path)) {
49                    sprintf(msg, "%s no existeix \n", path);
50                    fwrite(msg, strlen(msg), 1, stdout);
51                    continue;
52              }
53
54              sprintf(msg, "Type \t Size \t\t Epoc \t\t Name \n");
55              fwrite(msg, strlen(msg), 1, stdout);
56
57              if (how_many_params > 1) {
58                    sprintf(msg, "%s:\n", path);
59                    fwrite(msg, strlen(msg), 1, stdout);
60              }
61
62
63              if (emofs_is_file(path)) {
64                    /* ls d'un fitxer mostra sols el path del fitxer */
65                    emofs_stat(path, &stat);
66                    emofs_get_partial_path(path, partial_path, filename);
67                    sprintf(msg, "f \t %d \t\t %d \t %s \n",
68                          stat.size, stat.mtime, filename);
69                    fwrite(msg, strlen(msg), 1, stdout);
70              } else {
71                    entries_count = emofs_dir(path, &dir_content);
72                    for (j=0; j < entries_count; j++) {
73                          emofs_extract_dir_entry(dir_content, filename);
74                          /*Afegim path a l'element per poder fer el
75                           * emofs_stat */
```

```
76                          if (path[strlen(path)-1] != '/') {
77                                sprintf(full_path, "%s/%s", path, \
78                                      filename);
79                          } else {
80                                sprintf(full_path, "%s%s", path, \
81                                      filename);
82                          }
83                          emofs_stat(full_path, &stat);
84                          sprintf(msg, "%c \t %d \t\t %d \t %s \n", \
85                                ((stat.type==FILE_INODE) ? 'f' : 'd'), \
86                                stat.size, stat.mtime, filename);
87                          len = strlen(msg);
88                          fwrite(msg, len, 1, stdout);
89                    }
90                    free(dir_content);
91              }
92
93              if (how_many_params > 1) {
94                    sprintf(msg, "\n");
95                    fwrite(msg, strlen(msg), 1, stdout);
96              }
97        }
98
99        bumount();
100       return 0;
101 }
```

A.5.6 mi_mkdir.c

```
1  #include <stdlib.h>
2  #include <string.h>
3
4  #include "common.h"
5  #include "dir.h"
6
7  /*
8   * Crea un diretori.
9   * mi_mkdir <cami>
10  * @return: 0 exit, -1 nombre de parametres incorrecte
11  */
12 int main(int argc, char **argv) {
13       char path[MAX_PATH_LEN];
```

```
14        int error = 0;
15        int len;
16
17        if (argc != 2) {
18              puts("Nombre de parametres incorrecte.");
19              puts("Us: mi_mkdir <cami>");
20              return -1;
21        }
22        bmount();
23        strcpy(path, argv[1]);
24        len = strlen(path);
25        if (path[len-1] != '/') {
26              strcat(path, "/");
27        }
28        error = emofs_create(path);
29        if(error < 0) {
30              puts("mi_mkdir: el cami ja existeix");
31        }
32        bumount();
33        return error;
34
35  }
```

A.5.7 mi_rm.c

```
1   #include <stdlib.h>
2   #include <string.h>
3   #include <stdio.h>
4
5   #include "common.h"
6   #include "dir.h"
7
8   /* Esborra un directori i els seus subdirectoris i fitxers
9    * @dir_path: cami del directori a borrar
10   * @return 0 en cas d'exit.
11   * -1 en cas d'error.
12   */
13  int recursive_rm(char *path);
14
15  /*
16   * Esborra un fitxer. Si es un enllac simplement lleva
17   * l'enllac simbolic.
```

```
18  * mi_rm <cami>
19  * @return: 0 exit, -1 nombre de parametres incorrecte
20  */
21
22 int main(int argc, char **argv) {
23       char path[MAX_PATH_LEN];
24       int error = 0;
25
26       if (argc != 2) {
27             puts("Nombre de parametres incorrecte.");
28             puts("Us: mi_rm <cami>");
29             return -1;
30       }
31
32       bmount();
33       strcpy(path, argv[1]);
34       if (!emofs_file_exists(path)) {
35             printf("El fitxer/directori %s no existeix\n", path);
36             return -2;
37       }
38
39       recursive_rm(argv[1]);
40       bumount();
41       return error;
42 }
43
44 int recursive_rm(char *path) {
45       int i;
46       int entries_count;
47       char *dir_content; /* La funcio que l'omple ja reserva la memoria */
48       char filename[MAX_FILENAME_LEN];
49       char full_path[MAX_PATH_LEN];
50
51       if (emofs_is_file(path)) {
52             return emofs_unlink(path);
53       }
54
55       /* Borrarem subdirectoris */
56       entries_count = emofs_dir(path, &dir_content);
57       for (i=0; i < entries_count; i++) {
58             emofs_extract_dir_entry(dir_content, filename);
59             /* Afegim path a l'element per poder fer el
60              * emofs_stat */
```

```
61              sprintf(full_path, "%s/%s", path, filename);
62              recursive_rm(full_path);
63        }
64        free(dir_content);
65        /* Borram el propi directori */
66        return emofs_unlink(path);
67
68  }
```

A.5.8 mi_stat.c

```
1   #include <stdlib.h>
2   #include <string.h>
3   #include <stdio.h>
4
5   #include "common.h"
6   #include "dir.h"
7
8   /*
9    * Mostra informacio d'un fitxer
10   * mi_cat <nom>
11   * @return: 0 exit, -1 nombre de parametres incorrecte
12   */
13  int main(int argc, char **argv) {
14        char path[MAX_PATH_LEN];
15        char msg[7*80]; /* 7 linies de 80 caracters */
16        emofs_inode_stat info;
17
18        if (argc != 2) {
19              puts("Nombre de parametres incorrecte.");
20              puts("Us: mi_cat <nom>");
21              return -1;
22        }
23
24        strcpy(path, argv[1]);
25        bmount();
26
27        if (!emofs_file_exists(path)) {
28              puts("El fitxer o directori no existeix");
29              return -1;
30        }
31
```

```
32        emofs_stat(path, &info);
33        if (info.type == DIRECTORY_INODE) {
34              sprintf(msg, "Tipus: directori\n");
35        } else {
36              sprintf(msg, "Tipus: fitxer\n");
37        }
38        sprintf(msg, "%sTamany en bytes: %d\n", msg, info.size);
39        sprintf(msg, "%sData de modificacio: %d\n", msg, info.mtime);
40        sprintf(msg, "%sBlocs fisics asignats: %d\n", msg, info.block_count);
41        sprintf(msg, "%sNombre d'enllacos: %d\n", msg, info.link_count);
42        fwrite(msg, strlen(msg), 1, stdout);
43
44        bumount();
45
46        return 0;
47
48  }
```

A.5.9 mi_touch.c

```
1   #include <stdlib.h>
2   #include <time.h>
3   #include <string.h>
4
5   #include "common.h"
6   #include "dir.h"
7
8   /** Crea un fitxer buit
9    * mi_touch <data> <nom>
10   * Si s'indica la data s'ha de fer donat el format seguit de: yyyyMMddhhmmss
11   * @return: 0 exit, -1 nombre de parametres incorrecte
12   * @En un futur programar cas amb els parametres de segons
13   */
14  int main(int argc, char **argv) {
15        char path[MAX_PATH_LEN];
16        char data[15]; /* yyyymmddhhmmss */
17        int error = 0;
18        time_t new_time;
19
20        if ((argc == 1) || (argc >= 3)) {
21              puts("Nombre de parametres incorrecte.");
22              puts("Us: mi_touch <nom> <data>");
```

```
23              return -1;
24        }
25
26
27        strcpy(path, argv[1]);
28
29        bmount();
30
31        if (argc == 2) {
32              if (!emofs_file_exists(path)) {
33                    error = emofs_create(path);
34              }
35              error = emofs_update_time(path, time(NULL));
36        } else {
37              if (!emofs_file_exists(path)) {
38                    error = emofs_create(path);
39              }
40              strcpy(data, argv[2]); /* En un futur processar data */
41              error = emofs_update_time(path, new_time);
42        }
43        bumount();
44        return error;
45  }
```

A.5.10 mi_write.c

```
1   #include <stdlib.h>
2   #include <string.h>
3
4   #include "common.h"
5   #include "dir.h"
6
7   /** Escriu adins un fitxer.
8    * mi_write fitxer buffer <offset = 0>
9    * @fitxer: el fitxer on escriure
10   * @buffer: la cadena a escriure
11   * @offset: el desplacament. Per defecte 0.
12   * @return: 0 si exit
13   */
14  int main(int argc, char **argv) {
15        int offset = 0;
16        int error = 0;
```

```
17        if ((argc == 1) || (argc > 4)) {
18              puts("Nombre de parametres incorrecte.");
19              puts("Us: mi_write fitxer buffer <offset=0>");
20              return -1;
21        }
22        if (argc == 4) {
23              offset = atoi(argv[3]);
24        }
25        bmount();
26        /* argv[1] -> path */
27        if (!emofs_file_exists(argv[1])) {
28              error = emofs_create(argv[1]);
29        }
30        error = emofs_write(argv[1], argv[2], offset, strlen(argv[2]));
31        bumount();
32        return error > 0;
33  }
```

A.5.11 mi_mount.c

```
1   #include "sem.h"
2   /** Petit programa per inicialitzar els semafors. Cal cridar-lo abans d'usar
3    * qualsevol eina. */
4   int main() {
5         int mutex;
6         emofs_sem_init(&mutex);
7         return 0;
8   }
```

A.5.12 mi_umount.c

```
1   #include "sem.h"
2   /** Petit programa per destruir els semafors. No s'ha de cridar fins haver
3    * acabat totes les feines. */
4   int main() {
5         int mutex;
6         emofs_sem_get(&mutex);
7         emofs_sem_del(mutex);
8         return 0;
9   }
```

A.6 Jocs de proves

A.6.1 test_mapa_bits.c

```
#include <stdio.h>
#include "bitmap.h"

int main(void) {
	int i, zeros, uns;
	emofs_bitmap mapa = map_of_zero();

	mflip(&mapa);

	for(i = BLOCK_SIZE*8/2; i < BLOCK_SIZE*8; i++) {
		mwrite(i, 0, &mapa);
	}

	mflip(&mapa);

	for(i = 0; i < BLOCK_SIZE*8; i++) {
		if (mread(i, &mapa) == 0) {
			zeros++;
		} else {
			uns++;
		}
	}

	printf("hi ha 0: %d, 1: %d\n", zeros, uns);
	if (zeros == uns) {
		printf("Test passat correctament.\n");
	} else {
		printf("Error al test.\n");
	}

	return 0;
}
```

A.6.2 test_setmana1.c

```
# include <stdio.h>
# include <sys/time.h>
# include "block.h"
```

```
4
5  int main() {
6        int num_blocs = 100;
7        int i, j, bloc_aleatori;
8        emofs_block blck, blck_zero, blck_one, blck_0xee;
9        int f_fs;
10       int correcte = 1;
11       char nom_fs[80];
12       char buf;
13       struct timeval tv;
14       sprintf(nom_fs, EMOFS_IMAGE_FILE);
15
16       /* Test 0: Muntar i omplir de zeros */
17       printf("Tests de la setmana 1:\n\n");
18       printf("\t - Cream un sistema de fitxers dins el fitxer %s \
19 de %d blocs \n\t de %d bytes cada un.\n", nom_fs, num_blocs, BLOCK_SIZE);
20       f_fs = bmount();
21       printf("\t - Sistema muntat, ara **l'omplirem de 0s**.\n");
22       blck_zero = block_of_zero();
23       for (i = 0; i < num_blocs; i++) {
24             bwrite(i, &blck_zero);
25       }
26       for (i = 0; i < num_blocs; i++) {
27             bread(i, &blck);
28             for (j = 0; j < BLOCK_SIZE; j++){
29                   if (blck.valor[j] != 0x00) {
30                         printf("\t -t La posicio %d del bloc %d no son \
31 tot zeros si no %c (valor del char)\n", j, i, blck.valor[j]);
32                         correcte = 0;
33                   }
34             }
35       }
36       if (correcte == 1) {
37             printf("\t - Sembla que el sistema de fitxers s'ha omplert de \
38 zeros correctament\n");
39       }
40       bumount();
41       printf("\n\nDesmuntat sistema de fitxers, primera part del test \
42 passada correctament, ara pots mirar el fitxer (han de ser tot zeros).
   \
43 \nPitja qualsevol tecla per continuar.\n");
44       getc(stdin);
45
```

```
46        /* Test 1: Canviam els zeros dels blocs senars per uns */
47        f_fs = bmount(nom_fs);
48        blck_one = block_of_one();
49        for (i = 0; i < num_blocs; i++) {
50              if (i % 2 == 1) {
51                    bwrite(i, &blck_one);
52              }
53        }
54        for (i = 0; i < num_blocs; i++) {
55              bread(i, &blck);
56              if (i % 2 == 0) {
57              /* Han de ser zeros */
58                    for (j = 0; j < BLOCK_SIZE; j++){
59                          if (blck.valor[j] != 0x00) {
60                                printf("\t - La posicio %d del bloc \
61  %d no son tot zeros sino %c (valor del char)\n", j, i, blck.valor[j]);
62                                correcte = 0;
63                          }
64                    }
65              } else {
66                    for (j = 0; j < BLOCK_SIZE; j++){
67                          if (blck.valor[j] != 0xff) {
68                                printf("\t - La posicio %d del bloc \
69  %d no son tot uns sino %c (valor del char)\n", j, i, blck.valor[j]);
70                                correcte = 0;
71                          }
72                    }
73              }
74        }
75        if (correcte == 1) {
76              printf("\t - Sembla que el sistema de fitxers s'ha omplert de \
77  zeros i uns correctament\n");
78        }
79        bumount();
80        printf("\n\nDesmuntat sistema de fitxers, segona part del test \
81  passada correctament, ara pots mirar el fitxer (zeros i uns alternats).\
82  \nPitja qualsevol tecla per continuar.\n");
83        getc(stdin);
84
85        /* Test 2: Posam el valor 0xee a tots els bytes d'un block aleatori */
86        f_fs = bmount(nom_fs);
87
88        for (i = 0; i < BLOCK_SIZE; i++){
```

```
89              blck_0xee.valor[i] = 0xee;
90        }
91
92        gettimeofday(&tv,NULL);
93        srand(tv.tv_usec);
94        bloc_aleatori = rand() % num_blocs;
95
96        bwrite(bloc_aleatori, &blck_0xee);
97        bread(bloc_aleatori, &blck);
98
99        for(i = 0; i < BLOCK_SIZE && correcte; i++){
100             if (blck.valor[i] != blck_0xee.valor[i]) {
101                   correcte = 0;
102             }
103       }
104
105       if (correcte) {
106             printf("El bloc %d, s'ha omplert satifactoriament \
107 de \"0xee\". \n", bloc_aleatori);
108       } else {
109             printf("El bloc %d, *NO* s'ha omplert satifactoriament \
110 de \"0xee\". \n", bloc_aleatori);
111       }
112
113 }
```

A.6.3 test_setmana2.c

```
1  #include <time.h>
2  #include <stdio.h>
3
4  #include "bitmap.h"
5  #include "block.h"
6  #include "inode.h"
7  #include "super.h"
8
9  int main(int argc, char **argv) {
10       int i, nombre_blocs;
11       emofs_superblock sb;
12       int inici_mb, inici_ai, inici_bd;
13       int final_mb, final_ai, final_bd;
14
```

```
15      if (argc != 2) {
16      puts("Nombre de parametres incorrecte.");
17      puts("Us: tests_setmana2 <nom>");
18            return -1;
19      }
20
21      bmount(argv[1]);
22
23      printf("Tests setmana 2:\n");
24      printf("-Llegim el superblock en el seu ESTAT inicial\n");
25      sbread(&sb);
26      print_sb(&sb);
27
28      printf("Mostram els valors dels defines:\n");
29
30      printf("\t * super.h:\n");
31      printf("\t\t SUPERBLOCK_ITEMS = %d\n", SUPERBLOCK_ITEMS);
32      printf("\t\t SUPERBLOCK_SIZE = %d\n", SUPERBLOCK_SIZE);
33      printf("\t\t PADDING_BYTES = %d\n", PADDING_BYTES);
34
35      printf("\t * inode.h:\n");
36      printf("\t\t INODES_PER_BLOCK = %d\n", INODES_PER_BLOCK);
37      printf("\t\t INDIRECT_POINTERS_PER_BLOCK = %d\n", \
38            INDIRECT_POINTERS_PER_BLOCK);
39      printf("\t\t INODE_TARGET_SIZE = %d\n", INODE_TARGET_SIZE);
40      printf("\t\t INODE_ITEMS = %d\n", INODE_ITEMS);
41      printf("\t\t INDIRECT_POINTER_COUNT = %d\n", INDIRECT_POINTER_COUNT);
42      printf("\t\t INODE_MUST_SIZE = %d\n", INODE_MUST_SIZE);
43      printf("\t\t DIRECT_POINTER_COUNT = %d\n", DIRECT_POINTER_COUNT);
44
45      printf("\t * bitmap.h:\n");
46      printf("\t\t MAP_SIZE = %d\n", MAP_SIZE);
47
48      bumount();
49 }
```

A.6.4 test_setmana3.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "block.h"
4 #include "bitmap.h"
```

```
5  #include "inode.h"
6  #include "block_lib.h"
7
8  int comprovar(int n_inodes, int n_blocs);
9
10 int main(int argc, char **argv) {
11       int n, ni, nb, i;
12       int *inodes;
13       int *blocs;
14       emofs_superblock sb;
15       emofs_inode inode;
16
17       if (argc != 4) {
18       puts("Nombre de parametres incorrecte.");
19       puts("Us: tests_setmana3 <nom> <n inodes a reservar> \
20  <nombre blocs a reservar");
21             return -1;
22       }
23
24       ni = atoi(argv[2]);
25       nb = atoi(argv[3]);
26
27       printf("Iniciat el test sobre el fitxer %s, en un prinicipi es \
28 reserveran %d inodes i %d blocs.\n", argv[1], ni, nb);
29
30       bmount(argv[1]);
31
32       sbread(&sb);
33
34       inodes = malloc(sizeof(int)*ni);
35       blocs = malloc(sizeof(int)*nb);
36
37       /* Reservar ni inodes i nb blocks */
38       for (i = 0; i < ni; i++) {
39             inodes[i] = alloc_inode(FILE_INODE);
40             printf("Reservat inode %d, es el %d\n", i, inodes[i]);
41       }
42
43       for (i = 0; i < nb; i++) {
44             blocs[i] = alloc_block();
45             printf("Reservat el bloc %d, es el %d\n", i, blocs[i]);
46       }
47
```

```
48        /* Comprovar */
49        comprovar(ni, nb);
50
51
52        printf("Alliberam la primera meitat\n");
53        /* Alliberar ni/2 inodes i nb/2 blocks */
54        for (i = 0; i < ni/2; i++) {
55              free_inode(inodes[i]);
56        }
57
58        for (i = 0; i < nb/2; i++) {
59              free_block(blocs[i]);
60        }
61
62        /* Comprovar */
63        printf("Comprovam que hem alliberat be la primera meitat\n");
64        comprovar((ni/2 + (ni % 2)), (nb/2 + (nb % 2)));
65
66        /* Alliberar tots */
67        for (i = 0; i < sb.total_inode_count; i++) {
68              read_inode(i, &inode);
69              if (inode.type != FREE_INODE) {
70                    free_inode(i);
71              }
72        }
73
74        for (i = 0; i < sb.total_block_count; i++) {
75              if (bm_read(i) != 0) {
76                    free_block(i);
77              }
78        }
79
80        /* Comprovar */
81        comprovar(0, 0);
82
83        puts("Anam a reservar tot el que poguem");
84        /* Reservar tots */
85        for (i = 0; i < sb.total_inode_count; i++) {
86              alloc_inode(FILE_INODE);
87        }
88        puts("Reservats tots els inodes");
89        puts("Anam a reservar tots els blocs de dades");
90        for (i = 0; i < sb.total_data_block_count; i++) {
```

```
91              alloc_block();
92        }
93        puts("Reservats tots els blocs de dades");
94        /* Comprovar */
95        comprovar(sb.total_inode_count, sb.total_data_block_count);
96
97
98        puts("Tornam a alliberar-ho tot.");
99        /* Alliberar tots */
100       for (i = 0; i < sb.total_inode_count; i++) {
101             read_inode(i, &inode);
102             if (inode.type != FREE_INODE) {
103                   free_inode(i);
104             }
105       }
106
107       for (i = 0; i < sb.total_block_count; i++) {
108             if (bm_read(i) != 0) {
109                   free_block(i);
110             }
111       }
112
113       /* Comprovar */
114       comprovar(0, 0);
115
116       bumount();
117 }
118
119 int comprovar(int n_inodes, int n_blocs) {
120       int i, inodes_usats, blocs_usats;
121       emofs_superblock sb;
122       emofs_inode inode;
123
124       sbread(&sb);
125
126       printf("contador inodes: %d\n", sb.total_inode_count);
127       inodes_usats = 0;
128       for (i = 0; i < sb.total_inode_count; i++) {
129             read_inode(i, &inode);
130             if (inode.type != FREE_INODE) {
131                   inodes_usats++;
132             }
133       }
```

```
134
135         if (inodes_usats != n_inodes) {
136               printf("Error: S'havien de reservar %d inodes ", n_inodes);
137               printf("i se n'han reservat %d\n", inodes_usats);
138         } else {
139               printf("S'ha reservat el nombre d'inodes correcte.\n");
140         }
141
142         if ((sb.total_inode_count - sb.free_inode_count) != n_inodes ) {
143               printf("L'informacio del SB respecte els inodes ocupats es incorrecta"
144               printf("hi ha %d i haurien de ser %d\n",
145                (sb.total_inode_count - sb.free_inode_count), n_inodes);
146         } else {
147               printf("La informacio del SB respecte els inodes ocupats es correcta\n
148         }
149
150         blocs_usats = 0;
151         for (i = 0; i <= (sb.last_data_block - sb.first_data_block); i++) {
152               if (bm_read(i) == 1) {
153                     blocs_usats++;
154               }
155         }
156
157         if (blocs_usats != n_blocs) {
158               printf("Error: S'havien de reservar %d blocs ", n_blocs);
159               printf("i se n'han reservat %d\n", blocs_usats);
160         } else {
161               printf("S'ha reservat el nombre blocs correcte.\n");
162         }
163
164   }
```

A.6.5 test_setmana5.c

```
1   #include <stdio.h>
2   #include "block.h"
3
4   #define MAX_BLOCS_ALEATORIS 10
5
6   int main(int argc, char **argv) {
7         int ni;
8         int i, j;
```

```
9       int *inodes;
10      int *blocs_aleatoris[MAX_BLOCS_ALEATORIS];
11
12      if (argc != 2) {
13      puts("Nombre de parametres incorrecte.");
14      puts("Us: tests_setmana5 <n inodes a reservar>");
15            return -1;
16      }
17
18      ni = atoi(argv[1]);
19      inodes = malloc(sizeof(int)*ni);
20      blocs_aleatoris = malloc(sizeof(int)*MAX_BLOCS_ALEATORIS*ni);
21
22      /* Reservar un parell de inodes */
23      for (i = 0; i < ni; i++) {
24            inodes[i] = alloc_inode(FILE_INODE);
25            printf("Reservat inode %d, es el %d\n", i, inodes[i]);
26      }
27
28      /* assignar blocs de dades de manera aleatoria a punters
29       * de segon i tercer nivell (fitxer ralo) */
30      /* Alerta mirar que hi hagi suficients blocs al sistema de fitxers */
31      for (i = 0; i < ni; i++) {
32            for (j = 0, j < MAX_BLOCS_ALEATORIS; j++) {
33
34            }
35      }
36
37      /* Comprovar els ocupats mirant el mapa de bits */
38
39      /* Alliberar inodes */
40
41      /* Comprovar mapa de bits (ha de estar buit) */
42
43      /* Tornam a reservar exactament el mateix */
44      for (i = 0; i < ni; i++) {
45            inodes[i] = alloc_inode(FILE_INODE);
46            printf("Reservat inode %d, es el %d\n", i, inodes[i]);
47      }
48      for (i = 0; i < ni; i++) {
49            for (j = 0, j < MAX_BLOCS_ALEATORIS; j++) {
50
51            }
```

```
52          }
53
54          /* Truncam els fitxers per la meitat */
55
56          /* Comprovam que s'hi alliberat el que toca al mapa de bits
57           * i als inodes */
58
59          /* Truncam a 0 (borram) */
60
61          /* Tornam a Comprovar */
62
63
64  }
```

Índex