

---

# LINUX VFS

---

**Alexander Morales Panitz**  
Computer Scientist  
UTEC  
Lima, Peru  
alexander.morales@utec.edu.pe

**Grover Ugarte**  
Computer Scientist  
UTEC  
Lima, Peru  
grover.ugarte@utec.edu.pe

**Keywords** Virtual File System, Linux, LWNFS

## Implementación

La implementación del proyecto está almacenada en el siguiente repositorio:

**Código Fuente del proyecto:** <https://github.com/alexanderutec/Proyecto-Final-LWNFS>

## 1 Justificación de líneas modificadas

Para el presente proyecto en clase se proporcionó un código modificado de LWNFS soportado para las versiones superiores de Linux Kernel. A continuación, procederemos a analizar los cambios realizados para la compilación en la versión 6.4.8 Linux Kernel.

**NOTA:** La explicación se encuentra en los comentarios de las imágenes del código

```
1 /*
2- * Implementación del mkdir, touch y rmdir para lwnfs
3- * Proyecto final de Sistemas Operations - UTEC
4- * Integrantes: Morales Panitz, Alexander
5- * Ugarte Quispe, Grover
6 */
7 #include <linux/kernel.h>
8 #include <linux/init.h>
9 #include <linux/module.h>
10 #include <linux/pagemap.h> /* PAGE_CACHE_SIZE */
11 #include <linux/fs.h> /* This is where libfs stuff is declared */
12 #include <asm/atomic.h>
13 #include <asm/uaccess.h> /* copy_to_user */
14 /*
15 * Boilerplate stuff.
16 */
17 MODULE_LICENSE("GPL");
18 MODULE_AUTHOR("Jonathan Corbet");
19
20 #define LFS_MAGIC 0x19980122
21
22- /* Se declararon las variables globales al inicio del código
23- para evitar problemas de variables no definidas.*/
24-
25- // Se mantuvo la variable global counter.
26- static atomic_t counter, subcounter;
27- // Se declararon las variables globales para el manejo de archivos
28- struct inode_operations lwnfs_dir_inode_operations;
```

```
1 /*
2+ * Demonstrate a trivial filesystem using libfs.
3+ *
4+ * Copyright 2002, 2003 Jonathan Corbet <corbet@lwn.net>
5+ * This file may be redistributed under the terms of the GNU GPL.
6+ *
7+ * Chances are that this code will crash your system, delete your
8+ * nethack high scores, and set your disk drives on fire. You have
9+ * been warned.
10 */
11 #include <linux/kernel.h>
12 #include <linux/init.h>
13 #include <linux/module.h>
14 #include <linux/pagemap.h> /* PAGE_CACHE_SIZE */
15 #include <linux/fs.h> /* This is where libfs stuff is declared */
16 #include <asm/atomic.h>
17 #include <asm/uaccess.h> /* copy_to_user */
18 /*
19 * Boilerplate stuff.
20 */
21 MODULE_LICENSE("GPL");
22 MODULE_AUTHOR("Jonathan Corbet");
23
24 #define LFS_MAGIC 0x19980122
25
```

```

29
30 /*
31 * Anytime we make a file or directory in our filesystem we need to
32 * come up with an inode to represent it internally. This is
33 * the function that does that job. All that's really interesting
34 * is the "mode" parameter, which says whether this is a directory
35 * or file, and gives the permissions.
36 */
37
38 static struct inode *lfs_make_inode(struct super_block *sb, int mode)
39 {
40     struct inode *ret = new_inode(sb);
41     // Se especificó el kernel_time para el tiempo de creación del
42     // inode.
43     struct timespec64 kernel_time;
44     if (ret) {
45         ret->i_mode = mode;
46         // Cambio de sintaxis por versiones de Linux Kernel
47         ret->i_uid.val = ret->i_gid.val = 0;
48         ret->i_size = PAGE_SIZE;
49         ret->i_blocks = 0;
50         ret->i_atime = ret->i_mtime = ret->i_ctime = kernel_time;
51     }
52     return ret;
53 }
54 /*
55 * The operations on our "files".
56 */
57
58 /*
59 * Open a file. All we have to do here is to copy over a
60 * copy of the counter pointer so it's easier to get at.
61 */
62 static int lfs_open(struct inode *inode, struct file *filp)
63 {
64     // Cambio de sintaxis por versiones de Linux Kernel
65     filp->private_data = inode->i_private;
66     return 0;
67 }

```

```

68
69 #define THPSIZE 20
70 /*
71 * Read a file. Here we increment and read the counter, then pass it
72 * back to the caller. The increment only happens if the read is done
73 * at the beginning of the file (offset = 0); otherwise we end up
74 * counting
75 * by twos.
76 */
77 static ssize_t lfs_read_file(struct file *filp, char *buf,
78                             size_t count, loff_t *offset)
79 {
80     atomic_t *counter = (atomic_t *) filp->private_data;
81     int v, len;
82     char tmp[THPSIZE];
83     /*
84     * Encode the value, and figure out how much of it we can pass back.
85     */
86     v = atomic_read(counter);
87     if (*offset > 0)
88         v -= 1; /* the value returned when offset was zero */
89     else
90         atomic_inc(counter);
91     len = snprintf(tmp, THPSIZE, "%d\n", v);
92     if (*offset > len)
93         return 0;
94     if (count > len - *offset)
95         count = len - *offset;
96     /*
97     * Copy it back, increment the offset, and we're done.
98     */
99     if (copy_to_user(buf, tmp + *offset, count))
100         return -EFAULT;
101     *offset += count;
102     return count;
103 }
104 /*

```

```

26
27 /*
28 * Anytime we make a file or directory in our filesystem we need to
29 * come up with an inode to represent it internally. This is
30 * the function that does that job. All that's really interesting
31 * is the "mode" parameter, which says whether this is a directory
32 * or file, and gives the permissions.
33 */
34
35 static struct inode *lfs_make_inode(struct super_block *sb, int mode)
36 {
37     struct inode *ret = new_inode(sb);
38
39     if (ret) {
40         ret->i_mode = mode;
41         ret->i_uid = ret->i_gid = 0;
42         ret->i_blksize = PAGE_CACHE_SIZE;
43         ret->i_blocks = 0;
44         ret->i_atime = ret->i_mtime = ret->i_ctime = CURRENT_TIME;
45     }
46     return ret;
47 }
48 /*
49 * The operations on our "files".
50 */
51
52 /*
53 * Open a file. All we have to do here is to copy over a
54 * copy of the counter pointer so it's easier to get at.
55 */
56 static int lfs_open(struct inode *inode, struct file *filp)
57 {
58     filp->private_data = inode->i_private;
59     return 0;
60 }
61

```

```

62
63 #define THPSIZE 20
64 /*
65 * Read a file. Here we increment and read the counter, then pass it
66 * back to the caller. The increment only happens if the read is done
67 * at the beginning of the file (offset = 0); otherwise we end up
68 * counting
69 * by twos.
70 */
71 static ssize_t lfs_read_file(struct file *filp, char *buf,
72                             size_t count, loff_t *offset)
73 {
74     atomic_t *counter = (atomic_t *) filp->private_data;
75     int v, len;
76     char tmp[THPSIZE];
77     /*
78     * Encode the value, and figure out how much of it we can pass back.
79     */
80     v = atomic_read(counter);
81     if (*offset > 0)
82         v -= 1; /* the value returned when offset was zero */
83     else
84         atomic_inc(counter);
85     len = snprintf(tmp, THPSIZE, "%d\n", v);
86     if (*offset > len)
87         return 0;
88     if (count > len - *offset)
89         count = len - *offset;
90     /*
91     * Copy it back, increment the offset, and we're done.
92     */
93     if (copy_to_user(buf, tmp + *offset, count))
94         return -EFAULT;
95     *offset += count;
96     return count;
97 }
98 /*

```

```

104 /*
105  * Write a file.
106  */
107 static ssize_t lfs_write_file(struct file *filp, const char *buf,
108                             size_t count, loff_t *offset)
109 {
110     atomic_t *counter = (atomic_t *) filp->private_data;
111     char tmp[TMPSIZE];
112     /*
113      * Only write from the beginning.
114      */
115     if (*offset != 0)
116         return -EINVAL;
117     /*
118      * Read the value from the user.
119      */
120     if (count >= TMPSIZE)
121         return -EINVAL;
122     memset(tmp, 0, TMPSIZE);
123     if (copy_from_user(tmp, buf, count))
124         return -EFAULT;
125     /*
126      * Store it in the counter and we are done.
127      */
128     atomic_set(counter, simple_strtol(tmp, NULL, 10));
129     return count;
130 }
131
132
133 /*
134  * Now we can put together our file operations structure.
135  */
136 static struct file_operations lfs_file_ops = {
137     .open = lfs_open,
138     .read = lfs_read_file,
139     .write = lfs_write_file,
140 };
141
142
143 /*
144  * Create a file mapping a name to a counter.

```

```

98 /*
99  * Write a file.
100  */
101 static ssize_t lfs_write_file(struct file *filp, const char *buf,
102                             size_t count, loff_t *offset)
103 {
104     atomic_t *counter = (atomic_t *) filp->private_data;
105     char tmp[TMPSIZE];
106     /*
107      * Only write from the beginning.
108      */
109     if (*offset != 0)
110         return -EINVAL;
111     /*
112      * Read the value from the user.
113      */
114     if (count >= TMPSIZE)
115         return -EINVAL;
116     memset(tmp, 0, TMPSIZE);
117     if (copy_from_user(tmp, buf, count))
118         return -EFAULT;
119     /*
120      * Store it in the counter and we are done.
121      */
122     atomic_set(counter, simple_strtol(tmp, NULL, 10));
123     return count;
124 }
125
126
127 /*
128  * Now we can put together our file operations structure.
129  */
130 static struct file_operations lfs_file_ops = {
131     .open = lfs_open,
132     .read = lfs_read_file,
133     .write = lfs_write_file,
134 };
135
136
137 /*
138  * Create a file mapping a name to a counter.

```

```

145 /*
146 static struct dentry *lfs_create_file (struct super_block *sb,
147                                       struct dentry *dir, const char *name,
148                                       atomic_t *counter)
149 {
150     struct dentry *dentry;
151     struct inode *inode;
152     /* En la versión de linux kernel definida para el proyecto existe
153      * una función llamada d_alloc_name que permite crear una entrada de
154      * directorio con un nombre dado. Para la compilación del código base,
155      * este fue el problema principal */
156     /* Se asigna el nombre al directory entry */
157     dentry = d_alloc_name(dir, name);
158     if (! dentry)
159         goto out;
160     inode = lfs_make_inode(sb, S_IFREG | 0644);
161     if (! inode)
162         goto out_dput;
163     inode->i_fop = &lfs_file_ops;
164     /* Cambio de sintaxis por versiones de Linux Kernel */
165     inode->i_private = counter;
166     /*
167      * Put it all into the dentry cache and we're done.
168      */
169     d_add(dentry, inode);
170     return dentry;
171     /*
172      * Then again, maybe it didn't work.
173      */
174     out_dput:
175     dput(dentry);
176     out:
177     return 0;

```

```

139 /*
140 static struct dentry *lfs_create_file (struct super_block *sb,
141                                       struct dentry *dir, const char *name,
142                                       atomic_t *counter)
143 {
144     struct dentry *dentry;
145     struct inode *inode;
146     struct qstr qname;
147     /*
148      * Make a hashed version of the name to go with the dentry.
149      */
150     qname.name = name;
151     qname.len = strlen (name);
152     qname.hash = full_name_hash(name, qname.len);
153     /*
154      * Now we can create our dentry and the inode to go with it.
155      */
156     dentry = d_alloc(dir, &qname);
157     if (! dentry)
158         goto out;
159     inode = lfs_make_inode(sb, S_IFREG | 0644);
160     if (! inode)
161         goto out_dput;
162     inode->i_fop = &lfs_file_ops;
163     inode->u.generic_ip = counter;
164     /*
165      * Put it all into the dentry cache and we're done.
166      */
167     d_add(dentry, inode);
168     return dentry;
169     /*
170      * Then again, maybe it didn't work.
171      */
172     out_dput:
173     dput(dentry);
174     out:
175     return 0;

```

<pre> 177     return 0; 178 } 179 180 181 /* 182  * Create a directory which can be used to hold files. This code is 183  * almost identical to the "create file" logic, except that we create 184  * the inode with a different mode, and use the libfs "simple" 185  * operations. 186  */ 187 static struct dentry *lfs_create_dir (struct super_block *sb, 188                                     struct dentry *parent, const char *name) 189 { 190     struct dentry *dentry; 191     struct inode *inode; 192 193     /* En la versión de linux kernel definida para el proyecto existe 194     una función llamada d_alloc_name que permite crear una entrada de 195     directorio con un nombre dado. Para la compilación del código base, 196     este fue el problema principal */ 197 198     /* Se asigna el nombre al directory entry */ 199     dentry = d_alloc_name(parent, name); 200 201     if (!dentry) 202         goto out; 203 204     inode = lfs_make_inode(sb, S_IFDIR   0644); 205     if (!inode) 206         goto out_dput; 207 208     /* IMPORTANTE: Se cambió las operaciones por default para la 209     syscall de mkdir por las operaciones definidas en lwnfs. Esto 210     permita que se pueda realizar el llamo mkdir en cualquier 211     directorio padre */ 212     inode-&gt;i_op = &amp;lwnfs_dir_inode_operations; 213     inode-&gt;i_fop = &amp;simple_dir_operations; 214     d_add(dentry, inode); 215     return dentry; </pre>	<pre> 175     return 0; 176 } 177 178 179 /* 180  * Create a directory which can be used to hold files. This code is 181  * almost identical to the "create file" logic, except that we create 182  * the inode with a different mode, and use the libfs "simple" 183  * operations. 184  */ 185 static struct dentry *lfs_create_dir (struct super_block *sb, 186                                     struct dentry *parent, const char *name) 187 { 188     struct dentry *dentry; 189     struct inode *inode; 190     struct qstr qname; 191 192     qname.name = name; 193     qname.len = strlen(name); 194     qname.hash = full_name_hash(name, qname.len); 195     dentry = d_alloc(parent, &amp;qname); 196 197     if (!dentry) 198         goto out; 199 200     inode = lfs_make_inode(sb, S_IFDIR   0644); 201     if (!inode) 202         goto out_dput; 203     inode-&gt;i_op = &amp;simple_dir_inode_operations; 204 205     inode-&gt;i_fop = &amp;simple_dir_operations; 206     d_add(dentry, inode); 207     return dentry; </pre>
--	--

<pre> 208 209 out_dput: 210     dput(dentry); 211 out: 212     return 0; 213 } 214 215 /* A continuación, la implementación propia para el funcionamiento de 216 la syscall mkdir y touch. */ 217 218 /* Se procede a declarar la función de simple_mknod para la creación de 219 nodos en nuestro filesystem. */ 220 static int simple_mknod(struct inode *dir, struct dentry *dentry, 221                        umode_t mode, dev_t dev) 222 { 223     /* Ya que la función proporcionada por lwnfs para la creación 224     de archivos se requiere el contador. Para este caso usamos la 225     función atomic_inc() vista en el curso para realizar el 226     incremento cada vez que se cree un nuevo nodo */ 227     // Incrementamos el contador 228     atomic_inc(&amp;counter); 229     // Realizamos el llamado a la función create_file 230     dentry = lfs_create_file(dentry-&gt;d_sb, dentry-&gt;d_parent, 231                             dentry-&gt;d_name.name, &amp;counter); 232     return 0; 233 } 234 235 /* Se procede a declarar la función de simple_create para la creación 236 de nodos en nuestro filesystem. */ 237 static int simple_create(struct inode *inode, struct dentry *dentry, 238                        umode_t mode, bool excl) 239 { 240     /* Simplemente realizamos el llamado a la función simple_mknod para 241     la creación del nodo */ 242     return simple_mknod(inode, dentry, mode, 0); 243 } </pre>	<pre> 206 207 out_dput: 208     dput(dentry); 209 out: 210     return 0; 211 } 212 </pre>
---	---

```

240- /* Se procede a declarar la función de simple_mkdir para la creación de
- directorios en nuestro filesystem. */
241- static int simple_mkdir(struct inode *dir, struct dentry *dentry,
- umode_t mode)
242- {
243-     dentry = lfs_create_dir(dentry->d_sb, dentry->d_parent,
-     dentry->d_name.name);
244-     return 0;
245- }
246-
247- /* Tal como se había mencionado anteriormente, se requiere una propia
248- implementación de las funciones mkdir y touch. Por ello, definimos
249- las funciones implementadas en la variable global */
250-
251- /* IMPORTANTE: El linux kernel provee funciones simples para
252- las demás operaciones. Por esa razón, se hace el llamado a <linux/fs.
- h>
253- */
254-
255- struct inode_operations lwnfs_dir_inode_operations = {
256-     // Definimos la función de creación de archivos
257-     .create = simple_create,
258-     .lookup = simple_lookup,
259-     .link = simple_link,
260-     .unlink = simple_unlink,
261-     // Definimos la función de creación de directorios
262-     .mkdir = simple_mkdir,
263-     .rmdir = simple_rmdir,
264-     // Definimos la función de creación de nodos
265-     .mknod = simple_mknod,
266- };
267
268
269 /*
270  * OK, create the files that we export.
271  */

```

```

213
214
215 /*
216  * OK, create the files that we export.
217  */

```

```

272
273 static void lfs_create_files (struct super_block *sb, struct dentry
*root)
274 {
275     struct dentry *subdir;
276     /*
277      * One counter in the top-level directory.
278      */
279     atomic_set(&counter, 0);
280     lfs_create_file(sb, root, "counter", &counter);
281     /*
282      * And one in a subdirectory.
283      */
284     atomic_set(&subcounter, 0);
285     subdir = lfs_create_dir(sb, root, "subdir");
286     if (subdir)
287         lfs_create_file(sb, subdir, "subcounter", &subcounter);
288 }
289
290
291 /*
292  * Superblock stuff. This is all boilerplate to give the vfs something
293  * that looks like a filesystem to work with.
294  */
295
296 /*
297  * Our superblock operations, both of which are generic kernel ops
298  * that we don't have to write ourselves.
299  */
300 static struct super_operations lfs_s_ops = {
301     .statfs = simple_statfs,
302     .drop_inode = generic_delete_inode,

```

```

218+ static atomic_t counter, subcounter;
219
220 static void lfs_create_files (struct super_block *sb, struct dentry
*root)
221 {
222     struct dentry *subdir;
223     /*
224      * One counter in the top-level directory.
225      */
226     atomic_set(&counter, 0);
227     lfs_create_file(sb, root, "counter", &counter);
228     /*
229      * And one in a subdirectory.
230      */
231     atomic_set(&subcounter, 0);
232     subdir = lfs_create_dir(sb, root, "subdir");
233     if (subdir)
234         lfs_create_file(sb, subdir, "subcounter", &subcounter);
235 }
236
237
238 /*
239  * Superblock stuff. This is all boilerplate to give the vfs something
240  * that looks like a filesystem to work with.
241  */
242
243 /*
244  * Our superblock operations, both of which are generic kernel ops
245  * that we don't have to write ourselves.
246  */
247 static struct super_operations lfs_s_ops = {
248     .statfs = simple_statfs,
249     .drop_inode = generic_delete_inode,

```



```

303 };
304
305 /*
306  * "Fill" a superblock with mundane stuff.
307  */
308 static int lfs_fill_super (struct super_block *sb, void *data, int
silent)
309 {
310     struct inode *root;
311     struct dentry *root_dentry;
312     /*
313      * Basic parameters.
314      */
315     /* Cambio por la version de Linux Kernel */
316     sb->s_blocksize = PAGE_SIZE;
317     sb->s_blocksize_bits = PAGE_SHIFT;
318     sb->s_magic = LFS_MAGIC;
319     sb->s_op = &lfs_s_ops;
320     /*
321      * We need to conjure up an inode to represent the root directory
322      * of this filesystem. Its operations all come from libfs, so we
323      * don't have to mess with actually *doing* things inside this
324      * directory.
325      */
326     root = lfs_make_inode (sb, S_IFDIR | 0755);
327     if (! root)
328         goto out;
329     /* Definimos las operaciones de libfs para el directorio raíz */
330     root->i_op = &lwnfs_dir_inode_operations;
331     root->i_fop = &simple_dir_operations;
332     /*
333      * Get a dentry to represent the directory in core.
334      */
335     /* Cambio por la version de Linux Kernel */
336     root_dentry = d_make_root(root);
337     if (! root_dentry)
338         goto out_input;
339     sb->s_root = root_dentry;
340     /*
341      * Make up the files which will be in this filesystem, and we're done.
342      */
343     lfs_create_files (sb, root_dentry);
344     return 0;
345     out_input:
346         input(root);
347     out:
348         return -ENOMEM;
349 }
350
351 /*
352  * Stuff to pass in when registering the filesystem.
353  */
354 /* Cambio por la version de Linux Kernel */
355 static struct dentry *lfs_get_super(struct file_system_type *fst,
int flags, const char *devname, void *data)
356 {
357     /* Cambio por la version de Linux Kernel */
358     return mount_bdev(fst, flags, devname, data, lfs_fill_super);
359 }
360
361 static struct file_system_type lfs_type = {
362     .owner      = THIS_MODULE,
363     .name       = "lwnfs",
364     /* Cambio por la version de Linux Kernel */
365     .mount      = lfs_get_super,
366     .kill_sb    = kill_litter_super,
367 };
368
369
370
371
372
373 /*
374  * Get things set up.
375  */
376 static int __init lfs_init(void)
377 {
378     return register_filesystem(&lfs_type);
379 }
380
381 static void __exit lfs_exit(void)
382 {
383     unregister_filesystem(&lfs_type);
384 }
385
386 module_init(lfs_init);
387 module_exit(lfs_exit);

```

```

250 };
251
252 /*
253  * "Fill" a superblock with mundane stuff.
254  */
255 static int lfs_fill_super (struct super_block *sb, void *data, int
silent)
256 {
257     struct inode *root;
258     struct dentry *root_dentry;
259     /*
260      * Basic parameters.
261      */
262     sb->s_blocksize = PAGE_CACHE_SIZE;
263     sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
264     sb->s_magic = LFS_MAGIC;
265     sb->s_op = &lfs_s_ops;
266     /*
267      * We need to conjure up an inode to represent the root directory
268      * of this filesystem. Its operations all come from libfs, so we
269      * don't have to mess with actually *doing* things inside this
270      * directory.
271      */
272     root = lfs_make_inode (sb, S_IFDIR | 0755);
273     if (! root)
274         goto out;
275     root->i_op = &simple_dir_inode_operations;
276     root->i_fop = &simple_dir_operations;
277     /*
278      * Get a dentry to represent the directory in core.
279      */
280     root_dentry = d_alloc_root(root);
281     if (! root_dentry)
282         goto out_input;
283     sb->s_root = root_dentry;
284     /*
285      * Make up the files which will be in this filesystem, and we're done.
286      */
287     lfs_create_files (sb, root_dentry);
288     return 0;
289     out_input:
290         input(root);
291     out:
292         return -ENOMEM;
293 }
294
295 /*
296  * Stuff to pass in when registering the filesystem.
297  */
298 /* Cambio por la version de Linux Kernel */
299 static struct super_block *lfs_get_super(struct file_system_type *fst,
int flags, const char *devname, void *data)
300 {
301     return get_sb_single(fst, flags, data, lfs_fill_super);
302 }
303
304
305 static struct file_system_type lfs_type = {
306     .owner      = THIS_MODULE,
307     .name       = "lwnfs",
308     .get_sb     = lfs_get_super,
309     .kill_sb    = kill_litter_super,
310 };
311
312
313
314 /*
315  * Get things set up.
316  */
317 static int __init lfs_init(void)
318 {
319     return register_filesystem(&lfs_type);
320 }
321
322 static void __exit lfs_exit(void)
323 {
324     unregister_filesystem(&lfs_type);
325 }
326
327 module_init(lfs_init);
328 module_exit(lfs_exit);

```

## 2 Definir y montar Block Device

### 2.1 Block Device

Los dispositivos de bloque son dispositivos de hardware que se distinguen por lo aleatorio (es decir, no necesariamente secuencial) acceso de fragmentos de datos de tamaño fijo. Los fragmentos de datos de tamaño fijo se denominan bloques. El dispositivo de bloque más común es un disco duro, pero muchos otros dispositivos de bloque existen, como unidades de disquete, lectores de Blu-ray y memoria flash. Observe cómo estos son todos dispositivos en los que monta un sistema de archivos: los sistemas de archivos son la lengua franca de los bloques dispositivos (Linux Kernel Development, Robert Love).

### 2.2 Script para montar

```
# Para enlistar las particiones y los loops
$ sudo fdisk -l

# Revisar cual es el primer loop device habilitad. Por ejemplo /dev/loop11

# Definir nuestro directorio de montaje
$ mkdir ./mount_point

# Montar el block device en nuestro directorio
$ sudo mount -t lwnfs /dev/loop11 ./mount_point
```

### 3 Instalación sin INSMOD

#### 3.1 Explicación del procedimiento interno

Para explicar como se instala sin utilizar el comando **insmod <módulo>**, es necesario saber que procedimiento se realiza para la instalación de módulos.

El comando **insmod** hace una llamada al syscall **init\_module** el cual realiza un cargado de aquel archivo **.ko** listado junto al comando. Viendo detalladamente el syscall se aprecia lo siguiente:

```
int init_module(void *module_image, unsigned long len, const char *param_values);
```

- **module\_image**: el buffer binario del archivo *kernel object*
- **len**: el tamaño del archivo binario
- **param\_values**: vacío a menos de que se realice la carga del *kernel object* con valores específicos

Para facilitar el ingreso de parámetros en aquel syscall, existe una variación similar denominada **finit\_module**. Lo que hace es tomar en cuenta el identificador de un archivo en modo lectura cuando es llamada la syscall **int open(const char \*pathname, int flags);**.

```
int finit_module(int fd, const char *param_values, int flags);
```

- **fd**: el descriptor del archivo abierto en modo lectura
- **param\_values**: vacío a menos de que se realice la carga del *kernel object* con valores específicos
- **flags**: macros en caso se desee ignorar alguna característica de kernel del módulo

Para emplear aquellas syscalls en C, se menciona en la documentación que las funciones no están definidas. Por ese motivo, si se desea realizar un programa que llame a aquellas syscalls, es necesario hacer el llamado mediante la función **syscall()** (teniendo en cuenta de que el identificador de *finit\_module* se encuentra definido como el macro **\_\_NR\_finit\_module**).

```
syscall(__NR_finit_module, <fd>, <param_values>, <flags>);
```

#### 3.2 Desarrollo del programa de syscalls en C

Al momento de crear un programa en C que pueda realizar la instalación de módulos mediante syscalls solo con el listado del archivo como argumento de ejecución, es necesario tener en cuenta lo siguiente para los parámetros de la syscall:

- El nombre del archivo será especificado como argumentos de ejecución  

```
sudo ./instalar_modulo ejemplo.ko
```
- El módulo **lwnfs** no requiere algún parámetro adicional, por lo que **param\_values** será una cadena vacía
- El valor de **flags** será 0 debido a que no necesitamos ignorar algún componente específico de las características del módulo **lwnfs** compilado

##### instalar\_modulo.c

```
1 #define _GNU_SOURCE
2 #include <fcntl.h>
3 #include <stdio.h>
4 #include <sys/stat.h>
5 #include <sys/syscall.h>
6 #include <sys/types.h>
7 #include <unistd.h>
8 #include <stdlib.h>
9 #include <string.h>
10
11
```



```

12 long finit_module(int fd){
13     return syscall(__NR_finit_module, fd, "", 0);
14 }
15
16 int main(int argc, char **argv){
17
18     if(argc <= 1){
19         printf("ERROR: no hay algun archivo listado\nEJEMPLO: sudo ./
instalar_modulo archivo.ko\n");
20         return 0;
21     }
22     if(argc >= 3){
23         printf("ERROR: demasiados parametros\nEJEMPLO: sudo ./instalar_modulo
archivo.ko\n");
24         return 0;
25     }
26
27     int longitud = strlen(argv[1]);
28     if(longitud <= 3){
29         printf("ERROR: no es la longitud adecuada del nombre de un archivo .ko\
nEJEMPLO: sudo ./instalar_modulo archivo.ko\n");
30         return 0;
31     }
32     if(argv[1][longitud-3] != '.' || argv[1][longitud-2] != 'k' || argv[1][
longitud-1] != 'o'){
33         printf("ERROR: el archivo no tiene extension .ko\nEJEMPLO: sudo ./
instalar_modulo archivo.ko\n");
34         return 0;
35     }
36
37     int f_id = open(argv[1], O_RDONLY);
38
39     if(finit_module(f_id))
40         printf("FALLO LA INSTALACION, INTENTE OTRA VEZ (tal vez falten permisos)\
n");
41     else
42         printf("INSTALACION EXITOSA\n");
43
44     close(f_id);
45     return 0;
46 }

```

### 3.3 Instalación de módulos mediante syscalls

Se compila el archivo **instalar\_modulo.c** y se procede a ejecutar en superusuario con el módulo **lwnfs.ko** como parámetro.

```
sudo ./instalar_modulo lwnfs.ko
```

```

batman@kubuntu:~/lwnfs$ sudo make clean
rm -rf *.o *~ core .depend *.cmd *.ko *.mod.c *.symvers *.order
batman@kubuntu:~/lwnfs$ sudo make
make -C /lib/modules/5.4.161/build M=/home/batman/lwnfs modules
make[1]: Entering directory '/home/batman/Linux/linux-5.4.161'
Building with KERNELRELEASE = 5.4.161
  CC [M]  /home/batman/lwnfs/lwnfs.o
Building with KERNELRELEASE = 5.4.161
Building modules, stage 2.
MODPOST 1 modules
  CC [M]  /home/batman/lwnfs/lwnfs.mod.o
  LD [M]  /home/batman/lwnfs/lwnfs.ko
make[1]: Leaving directory '/home/batman/Linux/linux-5.4.161'
batman@kubuntu:~/lwnfs$ sudo ./instalar.out lwnfs.ko
INSTALACION EXITOSA
batman@kubuntu:~/lwnfs$ sudo mount -t lwnfs /dev/loop11 ./mount_point
batman@kubuntu:~/lwnfs$ sudo su
root@kubuntu:/home/batman/lwnfs# cd mount_point/
root@kubuntu:/home/batman/lwnfs/mount_point# ls
counter  subdir
root@kubuntu:/home/batman/lwnfs/mount_point# exit
exit
batman@kubuntu:~/lwnfs$ ls
compile.sh  instalar.c  lwnfs.c  lwnfs.mod  lwnfs.mod.o  Makefile  Module.symvers  mount.sh
desinstalar.c  instalar.out  lwnfs.ko  lwnfs.mod.c  lwnfs.o  modules.order  mount_point  umount.sh
batman@kubuntu:~/lwnfs$ # gcc instalar.c -o instalar.o
batman@kubuntu:~/lwnfs$ █

```

## 4 Pruebas de funcionamiento de filesystem

A continuación, se presentará la evidencia del funcionamiento del filesystem.

```
batman@kubuntu:~/lwnfs$ # Revisar el primer loop disponible
batman@kubuntu:~/lwnfs$ lsblk -l
NAME        MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
loop0        7:0      0    4K  1 loop /snap/bare/5
loop1        7:1      0   61,9M  1 loop /snap/core20/1405
loop2        7:2      0   161,5M  1 loop /snap/firefox/1498
loop3        7:3      0   61,9M  1 loop /snap/core20/1518
loop4        7:4      0   161,5M  1 loop /snap/firefox/1539
loop5        7:5      0   400,8M  1 loop /snap/gnome-3-38-2004/112
loop6        7:6      0   254,1M  1 loop /snap/gnome-3-38-2004/106
loop7        7:7      0    81,3M  1 loop /snap/gtk-common-themes/1534
loop8        7:8      0    47M   1 loop /snap/snapd/16010
loop9        7:9      0   91,7M  1 loop /snap/gtk-common-themes/1535
loop10       7:10     0    47M   1 loop /snap/snapd/16292
sda          8:0      0    60G   0 disk
sda1         8:1      0     1M   0 part
sda2         8:2      0   513M   0 part /boot/efi
sda3         8:3      0   59,5G   0 part /
batman@kubuntu:~/lwnfs$ # Notamos que puee ser el loop11
batman@kubuntu:~/lwnfs$ # Compilamos
batman@kubuntu:~/lwnfs$ sudo make clean && sudo make
rm -rf *.o ** core .depend *.cmd *.ko *.mod.c *.symvers *.order
make -C /lib/modules/5.4.161/build M=/home/batman/lwnfs modules
make[1]: Entering directory '/home/batman/Linux/linux-5.4.161'
Building with KERNELRELEASE = 5.4.161
CC [M] /home/batman/lwnfs/lwnfs.o
Building with KERNELRELEASE = 5.4.161
Building modules, stage 2.
MODPOST 1 modules
CC [M] /home/batman/lwnfs/lwnfs.mod.o
LD [M] /home/batman/lwnfs/lwnfs.ko
make[1]: Leaving directory '/home/batman/Linux/linux-5.4.161'
batman@kubuntu:~/lwnfs$ sudo insmod lwnfs.ko
batman@kubuntu:~/lwnfs$ # <- Instalamos con insmod
batman@kubuntu:~/lwnfs$ # Ahora, montamos
batman@kubuntu:~/lwnfs$ sudo mount -t lwnfs /dev/loop11 mount_point/
batman@kubuntu:~/lwnfs$ # Abrimos el root
batman@kubuntu:~/lwnfs$ sudo su
root@kubuntu:/home/batman/lwnfs# ls
compile.sh  instalar.c  lwnfs.ko    lwnfs.mod.c  lwnfs.o     modules.order  mount_point  umount.sh
desinstalar.c  lwnfs.c    lwnfs.mod  lwnfs.mod.o  Makefile    Module.symvers  mount.sh
root@kubuntu:/home/batman/lwnfs# # Nos vamos al directorio montado
root@kubuntu:/home/batman/lwnfs# cd mount_point/
root@kubuntu:/home/batman/lwnfs/mount_point# ls
counter  subdir
root@kubuntu:/home/batman/lwnfs/mount_point# # Creamos directorio
root@kubuntu:/home/batman/lwnfs/mount_point# mkdir newfiles
root@kubuntu:/home/batman/lwnfs/mount_point# ll
total 4
drwxr-xr-x 1 root root 4096 Sak 31 1969 ./
drwxrwxr-x 4 batman batman 4096 Kun 7 14:28 ../
-rw-r--r-- 1 root root 4096 Sak 31 1969 counter
drw-r--r-- 1 root root 4096 Sak 31 1969 newfiles/
drw-r--r-- 1 root root 4096 Sak 31 1969 subdir/
root@kubuntu:/home/batman/lwnfs/mount_point# cd newfiles
root@kubuntu:/home/batman/lwnfs/mount_point/newfiles# touch newfile
root@kubuntu:/home/batman/lwnfs/mount_point/newfiles# ll
total 0
drw-r--r-- 1 root root 4096 Sak 31 1969 ./
drwxr-xr-x 1 root root 4096 Sak 31 1969 ../
-rw-r--r-- 1 root root 4096 Kun 7 14:30 newfile
root@kubuntu:/home/batman/lwnfs/mount_point/newfiles# cat newfile
1
root@kubuntu:/home/batman/lwnfs/mount_point/newfiles# echo 100 > newfile
root@kubuntu:/home/batman/lwnfs/mount_point/newfiles# cat newfile
100
root@kubuntu:/home/batman/lwnfs/mount_point/newfiles#
```

## 5 Análisis de source code

A continuación, procederemos a definir las variables y las funciones generales del source code. **El análisis estará incluido en los comentarios** Como en todo programa, primero declaramos nuestras variables y funciones a utilizar

```

1
2 // Se mantuvo la variable global counter.
3 static atomic_t counter, subcounter;
4 // Se declararon las variables globales para el manejo de archivos
5 struct inode_operations lwnfs_dir_inode_operations;
6
7 // Funcion general para crear un nodo
8 static struct inode *lfs_make_inode(struct super_block *sb, int mode);
9
10 // Funcion para manejo de archivo open
11 static int lfs_open(struct inode *inode, struct file *filp);
12
13 // Funcion para manejo de archivo read
14 static ssize_t lfs_read_file(struct file *filp, char *buf,
15                             size_t count, loff_t *offset);
16
17 // Funcion para escribir en un archivo
18 static ssize_t lfs_write_file(struct file *filp, const char *buf,
19                              size_t count, loff_t *offset)
20 // FUncion para crear un archvio
21 static struct dentry *lfs_create_file (struct super_block *sb, struct dentry *dir
22                                     , const char *name, atomic_t *counter)
23
24 // Funcion para crear un directorio lwnfs
25 static struct dentry *lfs_create_dir (struct super_block *sb, struct dentry *
26                                     parent, const char *name)
27
28 // Sycall para crear un nodo
29 static int simple_mknod(struct inode *dir, struct dentry *dentry, umode_t mode,
30                        dev_t dev)
31
32 // Sycall para crear un archivo
33 static int simple_create(struct inode *inode, struct dentry *dentry, umode_t mode
34                        , bool excl)
35
36 // Sycall para crear un directorio
37 static int simple_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode)
38
39 // Funcion de ejemplo para crear los archivos
40 static void lfs_create_files (struct super_block *sb, struct dentry *root)
41
42 // Funcion para crear el directorio raiz
43 static int lfs_fill_super (struct super_block *sb, void *data, int silent)
44
45 // Funcion para inicializar el sistema de archivos
46 static struct dentry *lfs_get_super(struct file_system_type *fst, int flags,
47                                     const char *devname, void *data)
48
49 // Funcion para inicializar el sistema de archivos
50 static int __init lfs_init(void)
51
52 // Funcion para salir del sistema de archivos
53 static void __exit lfs_exit(void)

```

### 5.1 Creación de un nuevo nodo

```

1 static struct inode *lfs_make_inode(struct super_block *sb, int mode)
2 {
3     // Hacemos llamado a la funcion new_inode para

```

```

4 // inicializar un nuevo nodo
5 struct inode *ret = new_inode(sb);
6 // Se especifica el kernel_time para el tiempo de creacion del inode.
7 struct timespec64 kernel_time;
8
9 if (ret) {
10     // Definimos el modo
11     // Recordar que IF_REG es para archivos, IF_DIR es
12     // para directorios
13     ret->i_mode = mode;
14     // Cambio de sintaxis por versiones de Linux Kernel
15     ret->i_uid.val = ret->i_gid.val = 0;
16     // Tamanho del nodo
17     ret->i_size = PAGE_SIZE;
18     // Cantidad de bloques <- BTree
19     ret->i_blocks = 0;
20     // Anhadimos el tiempo al inode
21     ret->i_atime = ret->i_mtime = ret->i_ctime = kernel_time;
22 }
23 return ret;
24 }

```

## 5.2 Función para abrir un archivo

```

1 static int lfs_open(struct inode *inode, struct file *filp)
2 {
3     // Cambio de sintaxis por versiones de Linux Kernel
4     // Para cada archivo se extra el contador
5     // la cual esta almacenada en inode->i_private
6     filp->private_data = inode->i_private;
7     return 0;
8 }

```

## 5.3 Función para leer un archivo

```

1 static ssize_t lfs_read_file(struct file *filp, char *buf,
2     size_t count, loff_t *offset)
3 {
4     // Extraeemos el contador
5     atomic_t *counter = (atomic_t *) filp->private_data;
6     // Definimos las variables globales
7     int v, len;
8     char tmp[TMPSIZE];
9     // Leemos atomico <- Es un variable atomica
10    v = atomic_read(counter);
11    if (*offset > 0)
12        v -= 1; // el valor devuelto cuando el desplazamiento era cero
13    else
14        atomic_inc(counter);
15    len = snprintf(tmp, TMPSIZE, "%d\n", v);
16    if (*offset > len)
17        return 0;
18    if (count > len - *offset)
19        count = len - *offset;
20
21    // Copiamos e incrementamos
22    if (copy_to_user(buf, tmp + *offset, count))
23        return -EFAULT;
24    *offset += count;
25    return count;
26 }

```

## 5.4 Función para escribir un archivo

```

1  /*
2  * Write a file.
3  */
4  static ssize_t lfs_write_file(struct file *filp, const char *buf,
5                               size_t count, loff_t *offset)
6  {
7      // Extraemos el contador del archivo
8      atomic_t *counter = (atomic_t *) filp->private_data;
9      char tmp[TMPSIZE];
10
11     // Solo se puede escribir desde el inicio
12     if (*offset != 0)
13         return -EINVAL;
14     // Leer el valor por el usuario
15     if (count >= TMPSIZE)
16         return -EINVAL;
17     memset(tmp, 0, TMPSIZE);
18     if (copy_from_user(tmp, buf, count))
19         return -EFAULT;
20     // Guardar la informacion en el contador
21     atomic_set(counter, simple_strtol(tmp, NULL, 10));
22     return count;
23 }

```

## 5.5 Operaciones de un archivo

```

1  static struct file_operations lfs_file_ops = {
2      .open = lfs_open,
3      .read  = lfs_read_file,
4      .write = lfs_write_file,
5  };

```

## 5.6 Función general para crear un archivo

```

1  static int simple_create(struct inode *inode, struct dentry *dentry, umode_t mode
2                          , bool excl)
3  {
4      /* Simplemente realizamos el llamado a la funcion simple_mknod para
5       la creacion del nodo */
6      return simple_mknod(inode, dentry, mode, 0);

```

## 5.7 Función general para crear un directorio

```

1  static int simple_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode)
2  {
3      dentry = lfs_create_dir(dentry->d_sb, dentry->d_parent, dentry->d_name.name);
4      return 0;
5  }

```

## 5.8 Funcion para crear un nodo

```

1  static int simple_mknod(struct inode *dir, struct dentry *dentry, umode_t mode,
2                          dev_t dev)
3  {
4      /* Ya que la funcion proporcionada por lnfs para la creacion
5       de archivos se requiere el contador. Para este caso usamos la funcion
6       atomic_inc() vista en el curso para realizar el incremento cada vez que se
7       cree un nuevo nodo */

```



```

5 // Incrementamos el contador
6 atomic_inc(&counter);
7 // Realizamos el llamado a la funcion create_file
8 dentry = lfs_create_file(dentry->d_sb, dentry->d_parent, dentry->d_name.name,
9 &counter);
10 return 0;

```

## 5.9 Operaciones para el root principal

Según la investigación que se realizó, existen funciones generales como `siple_lookup`, `simple_link`, `simple_link`, y `simple_rmdir`. Cada filesystem tiene sus operaciones `simple_mknod`, `simple_create` y `simple_mkdir`. **Nuestra implementación soporta las operaciones `mkdir`, `touch` y `rmdir` como adicional.**

```

1 struct inode_operations lwnfs_dir_inode_operations = {
2 // Definimos la funcion de creacion de archivos
3 .create = simple_create,
4 .lookup = simple_lookup,
5 .link = simple_link,
6 .unlink = simple_unlink,
7 // Definimos la funcion de creacion de directorios
8 .mkdir = simple_mkdir,
9 .rmdir = simple_rmdir,
10 // Definimos la funcion de creacion de nodos
11 .mknod = simple_mknod,
12 };

```

## 5.10 Funcion de inicialización del super block y del directorio raiz

```

1 static void lfs_create_files (struct super_block *sb, struct dentry *root)
2 {
3 struct dentry *subdir;
4 // Setea como contador a 0
5 atomic_set(&counter, 0);
6 lfs_create_file(sb, root, "counter", &counter);
7 // Setea como subcontador a 0
8 atomic_set(&subcounter, 0);
9 subdir = lfs_create_dir(sb, root, "subdir");
10 if (subdir)
11 lfs_create_file(sb, subdir, "subcounter", &subcounter);
12 }

```

## 5.11 Funciones generales de inicializacion

Se declara las operaciones para el `super_block`

```

1 static struct super_operations lfs_s_ops = {
2 .statfs = simple_statfs,
3 .drop_inode = generic_delete_inode,
4 };

```

## 5.12 Registro del file system

Dicha función se encarga de inicializar los parametros del file system.

```

1 static int lfs_fill_super (struct super_block *sb, void *data, int silent)
2 {
3 struct inode *root;
4 struct dentry *root_dentry;
5 /*
6 * Basic parameters.
7 */

```

```

8      /* Cambio por la version de Linux Kernel */
9      sb->s_blocksize = PAGE_SIZE;
10     sb->s_blocksize_bits = PAGE_SHIFT;
11     sb->s_magic = LFS_MAGIC;
12     sb->s_op = &lfs_s_ops;
13 /*
14  * We need to conjure up an inode to represent the root directory
15  * of this filesystem. Its operations all come from libfs, so we
16  * don't have to mess with actually *doing* things inside this
17  * directory.
18  */
19     root = lfs_make_inode (sb, S_IFDIR | 0755);
20     if (! root)
21         goto out;
22     /* Definimos las operaciones de lwnfs para el directorio raiz */
23     root->i_op = &lwnfs_dir_inode_operations;
24     root->i_fop = &simple_dir_operations;
25 /*
26  * Get a dentry to represent the directory in core.
27  */
28     /* Cambio por la version de Linux Kernel */
29     root_dentry = d_make_root(root);
30     if (! root_dentry)
31         goto out_input;
32     sb->s_root = root_dentry;
33 /*
34  * Make up the files which will be in this filesystem, and we're done.
35  */
36     lfs_create_files (sb, root_dentry);
37     return 0;
38
39     out_input:
40     input(root);
41     out:
42     return -ENOMEM;
43 }

```

### 5.13 Funciones finales

Registro del file system, dar de baja el filesystem.

```

1  /*
2  * Get things set up.
3  */
4  static int __init lfs_init(void)
5  {
6      return register_filesystem(&lfs_type);
7  }
8
9  static void __exit lfs_exit(void)
10 {
11     unregister_filesystem(&lfs_type);
12 }
13
14 module_init(lfs_init);
15 module_exit(lfs_exit);

```

## 6 Explicar Linux VFS

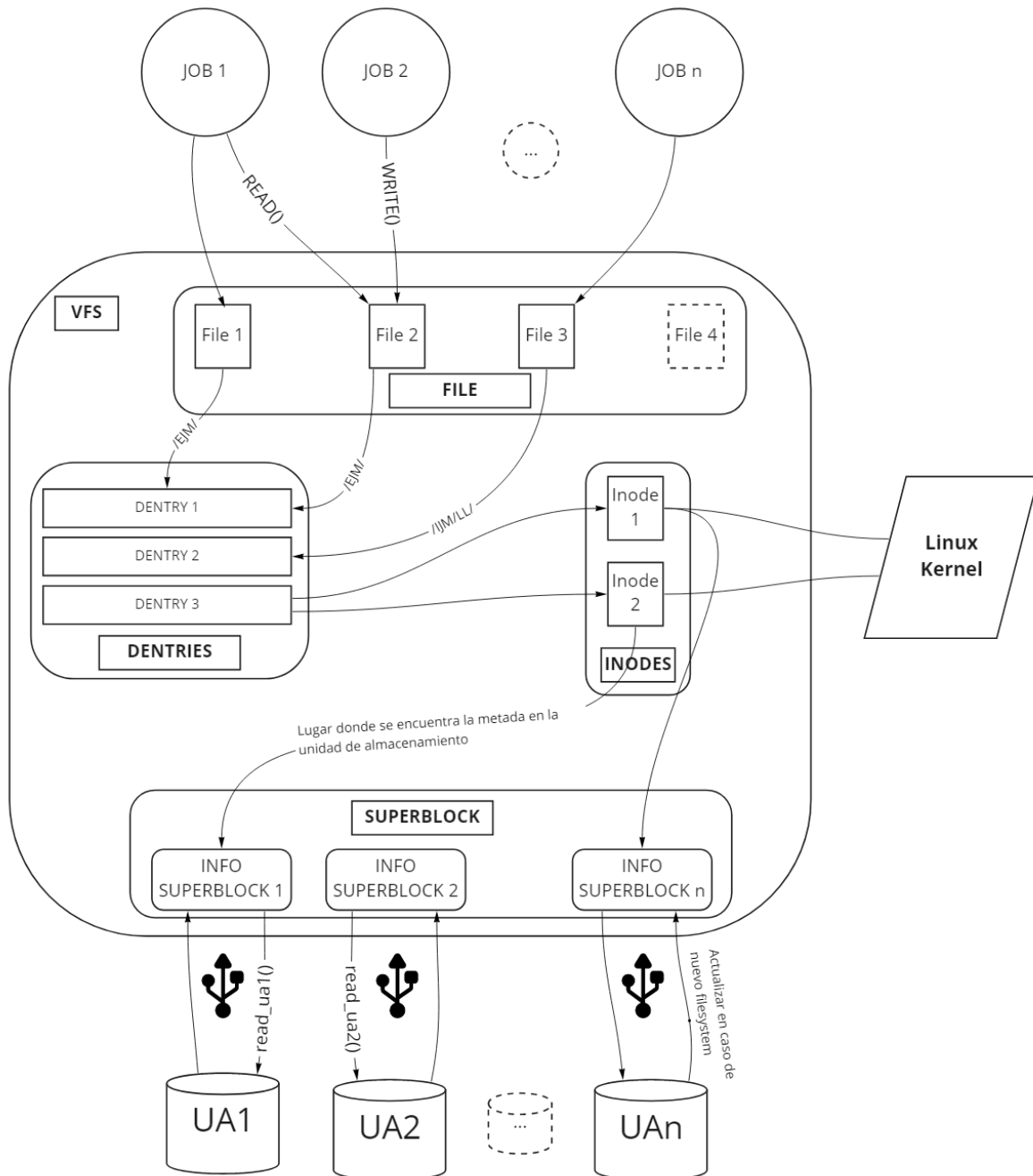
El sistema-de-archivos-virtual de Linux consiste en un ambiente abstracto virtual que permite que los programas puedan manipular cualquier sistema de archivos, y que aquellos sistemas de archivos puedan realizar operaciones entre sí. Aquello mencionado sin la necesidad de requerir algún cambio al código del programa o utilizando herramientas especiales.

Cuando se realiza la llamada a un syscall referido a archivos (read, write, open), lo que sucede es que el VFS procesa aquellas llamadas para redirigir a sus equivalentes dentro de aquel sistema de archivos que le pertenezca a la unidad de almacenamiento. Por ese motivo, nuevas unidades de almacenamiento con sistemas de archivos totalmente diferentes a los aplicados hasta aquel entonces pueden ser manipulados por los programas en Linux.

La forma en como el sistema-de-archivos-virtual maneja archivos y directorios es mediante 4 elementos fundamentales (objetos).

- Superbloque (superblock)  
Es un objeto que permite describir cada nuevo sistema de archivos (esta información se almacena y es extraída de un apartado específico del disco al momento de introducir un nuevo sistema de archivos al VFS).
- Nodos índice (inode)  
Es un objeto que acompaña a un archivo específico ya que contiene sus meta-datos de tal manera que el kernel tenga ciertas capacidades de trabajar con aquel. (No necesariamente todas las unidades de almacenamiento cuentan con esta estructura específica, sin embargo aquellos meta-datos pueden encontrarse en algún otro sector).
- Entrada de directorio (dentry) Es un objeto que toma en cuenta los elementos que conforman la ruta de un archivo específico. El VFS considera cada uno con un archivo y cada elemento apunta a un nodo índice cuya metadata indica como es que aquel archivo sería utilizado por el kernel.
- Archivo (file) Es el único objeto con el que cualquier tarea o proceso dentro del sistema operativo interactúa, principalmente leyendo o escribiendo. Se resalta de que cada objeto archivo señala a la entrada de directorio que pertenece.

Con esto en cuenta, podemos crear un diagrama que representa el funcionamiento de VFS.



Información obtenido de: **Linux Kernel Development, Robert Love.**

## References

- [1] `init_module(2)`: load kernel module - Linux man page. Retrieved from [https://linux.die.net/man/2/init\\_module](https://linux.die.net/man/2/init_module)
- [2] `syscall(2)` - Linux manual page. Retrieved from [https://man7.org/linux/man-pages/man2/syscall.2.html#:~:text=syscall\(\)%20is%20a%20small,function%20in%20the%20C%20library](https://man7.org/linux/man-pages/man2/syscall.2.html#:~:text=syscall()%20is%20a%20small,function%20in%20the%20C%20library).
- [3] `open(2)` - Linux manual page. Retrieved from <https://man7.org/linux/man-pages/man2/open.2.html>
- [4] Linux Loadable Kernel Module HOWTO. Retrieved from <https://tldp.org/HOWTO/Module-HOWTO/x627.html>
- [5] Robert Love. (2010). *Linux Kernel Development* (3rd ed., pp. 261-288). Crawfordsville, Indiana: Pearson.