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

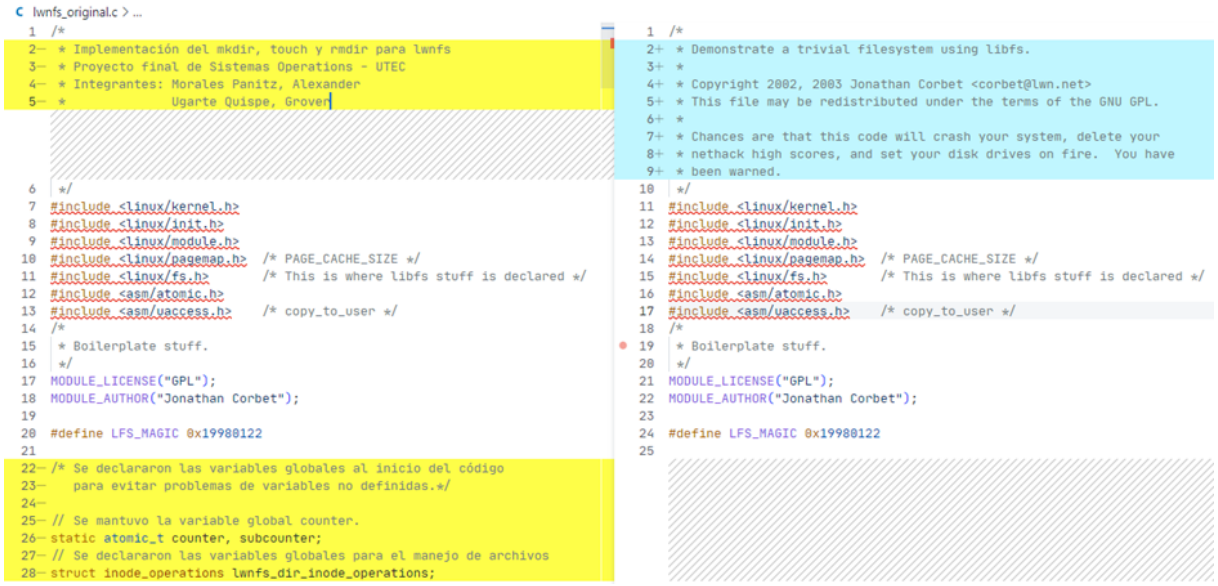
***K*eywords** Virtual File System, Linux, LWNFS

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



```
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 especifició el kernel_time para el tiempo de creacion del
    -    inode.
42-    struct timespec64 kernel_time;
43     if (ret) {
44         ret->i_mode = mode;
45-        // Cambio de sintaxis por versiones de Linux Kernel
46-        ret->i_uid.val = ret->i_gid.val = 0;
47-        ret->i_size = PAGE_SIZE;
48         ret->i_blocks = 0;
49-        ret->i_atime = ret->i_mtime = ret->i_ctime = kernel_time;
50     }
51     return ret;
52 }
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 }
```

```
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 /*
50  * The operations on our "files".
51  */
52
53 /*
54  * Open a file.  All we have to do here is to copy over a
55  * copy of the counter pointer so it's easier to get at.
56  */
57 static int lfs_open(struct inode *inode, struct file *filp)
58 {
59+    filp->private_data = inode->u.generic_ip;
60     return 0;
61 }
```

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

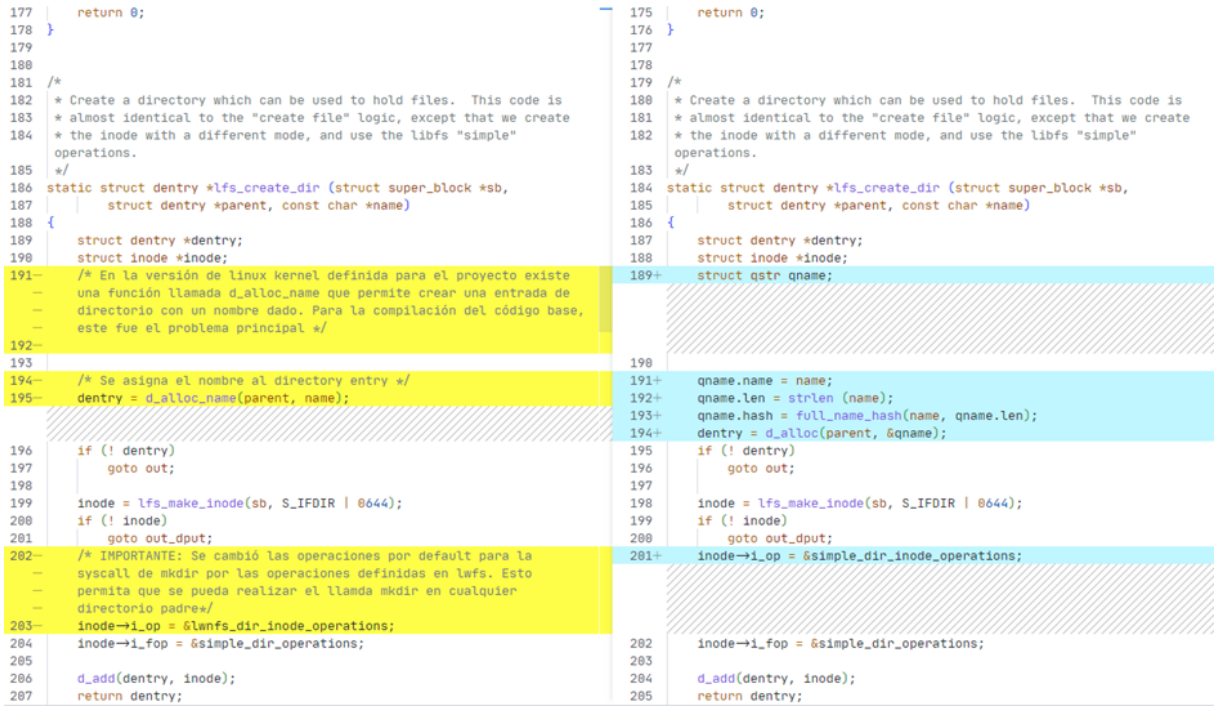
```
62
63 #define TMPSIZE 20
64 /*
65  * Read a file.  Here we increment and read the counter, then pass it
66  * back to the caller.  The increment only happens if the read is done
67  * at the beginning of the file (offset = 0); otherwise we end up
    counting
68  * by twos.
69  */
70 static ssize_t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

```
/*
 * Write a file.
 */
static ssize_t lfs_write_file(struct file *filp, const char *buf,
        size_t count, loff_t *offset)
{
    atomic_t *counter = (atomic_t *) filp->private_data;
    char tmp[TMPSIZE];
/*
 * Only write from the beginning.
 */
    if (*offset != 0)
        return -EINVAL;
/*
 * Read the value from the user.
 */
    if (count >= TMPSIZE)
        return -EINVAL;
    memset(tmp, 0, TMPSIZE);
    if (copy_from_user(tmp, buf, count))
        return -EFAULT;
/*
 * Store it in the counter and we are done.
 */
    atomic_set(counter, simple_strtol(tmp, NULL, 10));
    return count;
}


/*
 * Now we can put together our file operations structure.
 */
static struct file_operations lfs_file_ops = {
    .open   = lfs_open,
    .read   = lfs_read_file,
    .write  = lfs_write_file,
};


/*
 * Create a file mapping a name to a counter.
```

```
/*
 * Write a file.
 */
static ssize_t lfs_write_file(struct file *filp, const char *buf,
        size_t count, loff_t *offset)
{
    atomic_t *counter = (atomic_t *) filp->private_data;
    char tmp[TMPSIZE];
/*
 * Only write from the beginning.
 */
    if (*offset != 0)
        return -EINVAL;
/*
 * Read the value from the user.
 */
    if (count >= TMPSIZE)
        return -EINVAL;
    memset(tmp, 0, TMPSIZE);
    if (copy_from_user(tmp, buf, count))
        return -EFAULT;
/*
 * Store it in the counter and we are done.
 */
    atomic_set(counter, simple_strtol(tmp, NULL, 10));
    return count;
}


/*
 * Now we can put together our file operations structure.
 */
static struct file_operations lfs_file_ops = {
    .open   = lfs_open,
    .read   = lfs_read_file,
    .write  = lfs_write_file,
};


/*
 * Create a file mapping a name to a counter.
```

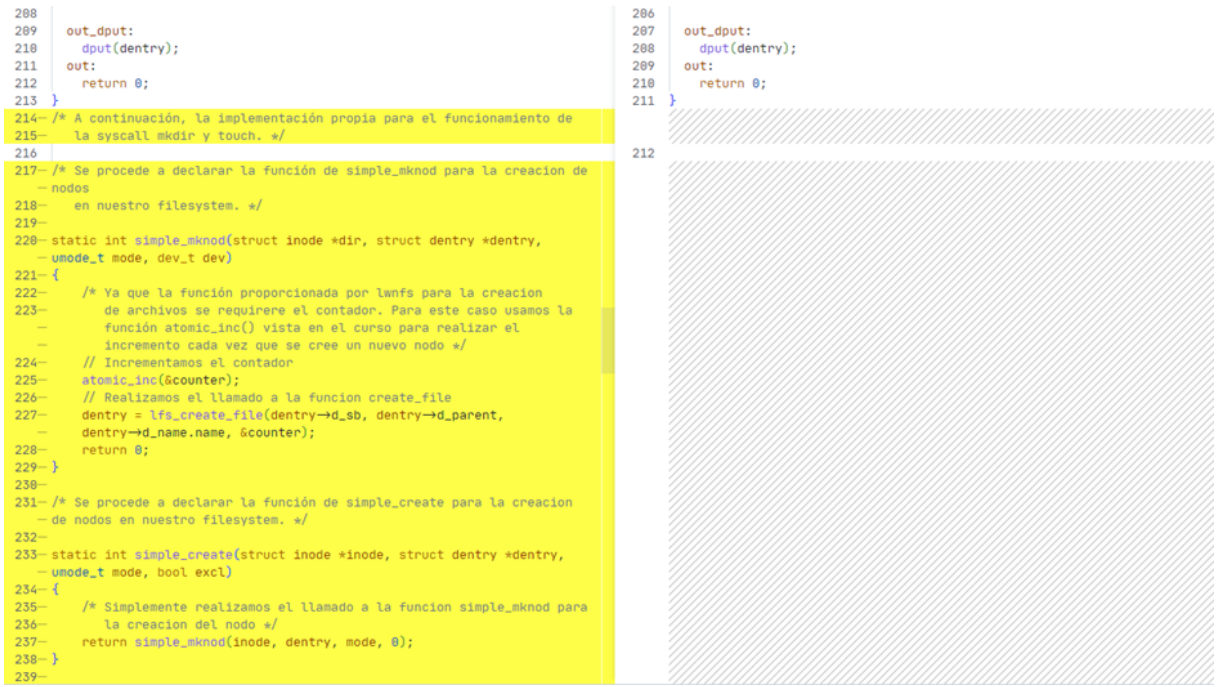
```
 */
static struct dentry *lfs_create_file (struct super_block *sb,
        struct dentry *dir, const char *name,
        atomic_t *counter)
{
    struct dentry *dentry;
    struct inode *inode;
    /* En la versión de linux kernel definida para el proyecto existe
    una función llamada d_alloc_name que permite crear una entrada de
    directorio con un nombre dado. Para la compilación del código base,
    este fue el problema principal */


/*
 * Now we can create our dentry and the inode to go with it.
 */
    /* Se asigna el nombre al directory entry */
    dentry = d_alloc_name(dir, name);
    if (! dentry)
        goto out;
    inode = lfs_make_inode(sb, S_IFREG | 0644);
    if (! inode)
        goto out_dput;
    inode->i_fop = &lfs_file_ops;
    /* Cambio de sintaxis por versiones de Linux Kernel */
    inode->i_private = counter;
/*
 * Put it all into the dentry cache and we're done.
 */
    d_add(dentry, inode);
    return dentry;
/*
 * Then again, maybe it didn't work.
 */
 out_dput:
    dput(dentry);
 out:
    return 0;
```

```
 */
static struct dentry *lfs_create_file (struct super_block *sb,
        struct dentry *dir, const char *name,
        atomic_t *counter)
{
    struct dentry *dentry;
    struct inode *inode;
    struct qstr qname;
/*
 * Make a hashed version of the name to go with the dentry.
 */
    qname.name = name;
    qname.len = strlen (name);
    qname.hash = full_name_hash(name, qname.len);
/*
 * Now we can create our dentry and the inode to go with it.
 */
    dentry = d_alloc(dir, &qname);

    if (! dentry)
        goto out;
    inode = lfs_make_inode(sb, S_IFREG | 0644);
    if (! inode)
        goto out_dput;
    inode->i_fop = &lfs_file_ops;
    inode->u.generic_ip = counter;

/*
 * Put it all into the dentry cache and we're done.
 */
    d_add(dentry, inode);
    return dentry;
/*
 * Then again, maybe it didn't work.
 */
 out_dput:
    dput(dentry);
 out:
    return 0;
```



```
    return 0;
}


/*
 * Create a directory which can be used to hold files.  This code is
 * almost identical to the "create file" logic, except that we create
 * the inode with a different mode, and use the libfs "simple"
 operations.
 */
static struct dentry *lfs_create_dir (struct super_block *sb,
        struct dentry *parent, const char *name)
{
    struct dentry *dentry;
    struct inode *inode;
    /* En la versión de linux kernel definida para el proyecto existe
    una función llamada d_alloc_name que permite crear una entrada de
    directorio con un nombre dado. Para la compilación del código base,
    este fue el problema principal */

    /* Se asigna el nombre al directory entry */
    dentry = d_alloc_name(parent, name);


    if (! dentry)
        goto out;

    inode = lfs_make_inode(sb, S_IFDIR | 0644);
    if (! inode)
        goto out_dput;
    /* IMPORTANTE: Se cambió las operaciones por default para la
    syscall de mkdir por las operaciones definidas en lwfs. Esto
    permita que se pueda realizar el llamda mkdir en cualquier
    directorio padre*/
    inode->i_op = &lwnfs_dir_inode_operations;
    inode->i_fop = &simple_dir_operations;

    d_add(dentry, inode);
    return dentry;
```

```
    return 0;
}


/*
 * Create a directory which can be used to hold files.  This code is
 * almost identical to the "create file" logic, except that we create
 * the inode with a different mode, and use the libfs "simple"
 operations.
 */
static struct dentry *lfs_create_dir (struct super_block *sb,
        struct dentry *parent, const char *name)
{
    struct dentry *dentry;
    struct inode *inode;
    struct qstr qname;




    qname.name = name;
    qname.len = strlen (name);
    qname.hash = full_name_hash(name, qname.len);
    dentry = d_alloc(parent, &qname);
    if (! dentry)
        goto out;

    inode = lfs_make_inode(sb, S_IFDIR | 0644);
    if (! inode)
        goto out_dput;
    inode->i_op = &simple_dir_inode_operations;




    inode->i_fop = &simple_dir_operations;

    d_add(dentry, inode);
    return dentry;
```



```

  out_dput:
    dput(dentry);
  out:
    return 0;
}
/* A continuación, la implementación propia para el funcionamiento de
   la syscall mkdir y touch. */

/* Se procede a declarar la función de simple_mknod para la creacion de
nodos
   en nuestro filesystem. */

static int simple_mknod(struct inode *dir, struct dentry *dentry,
umode_t mode, dev_t dev)
{
    /* Ya que la función proporcionada por lwnfs para la creacion
       de archivos se requirere el contador. Para este caso usamos la
       función atomic_inc() vista en el curso para realizar el
       incremento cada vez que se cree un nuevo nodo */
    // Incrementamos el contador
    atomic_inc(&counter);
    // Realizamos el llamado a la funcion create_file
    dentry = lfs_create_file(dentry->d_sb, dentry->d_parent,
    dentry->d_name.name, &counter);
    return 0;
}

/* Se procede a declarar la función de simple_create para la creacion
de nodos en nuestro filesystem. */

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

```

```

  out_dput:
    dput(dentry);
  out:
    return 0;
}


```

```
240- /* Se procede a declarar la función de simple_mkdir para la creacion 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 habia mencionado anteriormente, se require una propia
248-     implementacion 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 demas 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 funcion de creacion de archivos
257-     .create = simple_create,
258-     .lookup = simple_lookup,
259-     .link = simple_link,
260-     .unlink = simple_unlink,
261-     // Definimos la funcion de creacion de directorios
262-     .mkdir = simple_mkdir,
263-     .rmdir = simple_rmdir,
264-     // Definimos la funcion de creacion 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,
```

```
};

/*
 * "Fill" a superblock with mundane stuff.
 */
static int lfs_fill_super (struct super_block *sb, void *data, int
silent)
{
    struct inode *root;
    struct dentry *root_dentry;
/*
 * Basic parameters.
 */
-   /* Cambio por la version de Linux Kernel */
-   sb->s_blocksize = PAGE_SIZE;
-   sb->s_blocksize_bits = PAGE_SHIFT;
    sb->s_magic = LFS_MAGIC;
    sb->s_op = &lfs_s_ops;
/*
 * We need to conjure up an inode to represent the root directory
 * of this filesystem.  Its operations all come from libfs, so we
 * don't have to mess with actually *doing* things inside this
 * directory.
 */
    root = lfs_make_inode (sb, S_IFDIR | 0755);
    if (! root)
        goto out;
-   /* Definimos las operaciones de lwnfs para el directorio raiz */
-   root->i_op = &lwnfs_dir_inode_operations;
    root->i_fop = &simple_dir_operations;
/*
 * Get a dentry to represent the directory in core.
 */
-   /* Cambio por la version de Linux Kernel */
-   root_dentry = d_make_root(root);
    if (! root_dentry)
        goto out_iput;
    sb->s_root = root_dentry;
/*
 * Make up the files which will be in this filesystem, and we're done.
 */
    lfs_create_files (sb, root_dentry);
    return 0;

 out_iput:
    iput(root);
 out:
    return -ENOMEM;
}


/*
 * Stuff to pass in when registering the filesystem.
 */
-/* Cambio por la version de Linux Kernel */
-static struct dentry *lfs_get_super(struct file_system_type *fst,
        int flags, const char *devname, void *data)
{
-   /* Cambio por la version de Linux Kernel */
-   return mount_bdev(fst, flags, devname, data, lfs_fill_super);
}

static struct file_system_type lfs_type = {
    .owner      = THIS_MODULE,
    .name       = "lwnfs",
-   /* Cambio por la version de Linux Kernel */
-   .mount      = lfs_get_super,
    .kill_sb    = kill_litter_super,
};


/*
 * Get things set up.
 */
static int __init lfs_init(void)
{
    return register_filesystem(&lfs_type);
}

static void __exit lfs_exit(void)
{
    unregister_filesystem(&lfs_type);
}

module_init(lfs_init);
module_exit(lfs_exit);
```

```
};

/*
 * "Fill" a superblock with mundane stuff.
 */
static int lfs_fill_super (struct super_block *sb, void *data, int
silent)
{
    struct inode *root;
    struct dentry *root_dentry;
/*
 * Basic parameters.
 */
+   sb->s_blocksize = PAGE_CACHE_SIZE;
+   sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
    sb->s_magic = LFS_MAGIC;
    sb->s_op = &lfs_s_ops;
/*
 * We need to conjure up an inode to represent the root directory
 * of this filesystem.  Its operations all come from libfs, so we
 * don't have to mess with actually *doing* things inside this
 * directory.
 */
    root = lfs_make_inode (sb, S_IFDIR | 0755);
    if (! root)
        goto out;
+   root->i_op = &simple_dir_inode_operations;
    root->i_fop = &simple_dir_operations;
/*
 * Get a dentry to represent the directory in core.
 */
+   root_dentry = d_alloc_root(root);
    if (! root_dentry)
        goto out_iput;
    sb->s_root = root_dentry;
/*
 * Make up the files which will be in this filesystem, and we're done.
 */
    lfs_create_files (sb, root_dentry);
    return 0;

 out_iput:
    iput(root);
 out:
    return -ENOMEM;
}


/*
 * Stuff to pass in when registering the filesystem.
 */
+static struct super_block *lfs_get_super(struct file_system_type *fst,
        int flags, const char *devname, void *data)
{
+   return get_sb_single(fst, flags, data, lfs_fill_super);
}

static struct file_system_type lfs_type = {
    .owner      = THIS_MODULE,
    .name       = "lwnfs",
+   .get_sb     = lfs_get_super,
    .kill_sb    = kill_litter_super,
};


/*
 * Get things set up.
 */
static int __init lfs_init(void)
{
    return register_filesystem(&lfs_type);
}

static void __exit lfs_exit(void)
{
    unregister_filesystem(&lfs_type);
}

module_init(lfs_init);
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 <modulo>**, 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
```

```c
long finit_module(int fd){
    return syscall(__NR_finit_module, fd, "", 0);
}

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

    if(argc <= 1){
        printf("ERROR: no hay algun archivo listado\nEJEMPLO: sudo ./instalar_modulo archivo.ko\n");
        return 0;
    }
    if(argc >= 3){
        printf("ERROR: demasiados parametros\nEJEMPLO: sudo ./instalar_modulo archivo.ko\n");
        return 0;
    }

    int longitud = strlen(argv[1]);
    if(longitud <= 3){
        printf("ERROR: no es la longitud adecuada del nombre de un archivo .ko\nEJEMPLO: sudo ./instalar_modulo archivo.ko\n");
        return 0;
    }
    if(argv[1][longitud-3] != '.' || argv[1][longitud-2] != 'k' || argv[1][longitud-1] != 'o'){
        printf("ERROR: el archivo no tiene extension .ko\nEJEMPLO: sudo ./instalar_modulo archivo.ko\n");
        return 0;
    }

    int f_id = open(argv[1], O_RDONLY);

    if(finit_module(f_id))
        printf("FALLO LA INSTALACION, INTENTE OTRA VEZ (tal vez falten permisos)\n");
    else
        printf("INSTALACION EXITOSA\n");

    close(f_id);
    return 0;
}
```

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$ # <- Instalmos 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
       , const char *name, atomic_t *counter)
22
23 // Funcion para crear un directorio lwnfs
24 static struct dentry *lfs_create_dir (struct super_block *sb, struct dentry *
       parent, const char *name)
25
26 // Sycall para crear un nodo
27 static int simple_mknod(struct inode *dir, struct dentry *dentry, umode_t mode,
       dev_t dev)
28
29 // Sycall para crear un archivo
30 static int simple_create(struct inode *inode, struct dentry *dentry, umode_t mode
       , bool excl)
31
32 // Sycall para crear un directorio
33 static int simple_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode)
34
35 // Funcion de ejemplo para crear los archivos
36 static void lfs_create_files (struct super_block *sb, struct dentry *root)
37
38 // Funcion para crear el directorio raiz
39 static int lfs_fill_super (struct super_block *sb, void *data, int silent)
40
41 // Funcion para inicializar el sistema de archivos
42 static struct dentry *lfs_get_super(struct file_system_type *fst, int flags,
       const char *devname, void *data)
43
44 // Funcion para inicializar el sistema de archivos
45 static int __init lfs_init(void)
46
47 // Funcion para salir del sistema de archivos
48 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
```

```
    // inicializar un nuevo nodo
  struct inode *ret = new_inode(sb);
    // Se especificio el kernel_time para el tiempo de creacion del inode.
  struct timespec64 kernel_time;

    if (ret) {
        // Definimos el modo
        // Recordar que IF_REG es para archivos, IF_DIR es
        // para directorios
      ret->i_mode = mode;
        // Cambio de sintaxis por versiones de Linux Kernel
    ret->i_uid.val = ret->i_gid.val = 0;
        // Tamanhoo del nodo
    ret->i_size = PAGE_SIZE;
        // Cantidad e bloques <- BTree
    ret->i_blocks = 0;
        // Anhadimos el tiempo al inode
    ret->i_atime = ret->i_mtime = ret->i_ctime = kernel_time;
  }
  return ret;
}
```

5.2 Función para abrir un archivo

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

5.3 Función para leer un archivo

```
static ssize_t lfs_read_file(struct file *filp, char *buf,
    size_t count, loff_t *offset)
{
    // Extraeemos el contador
  atomic_t *counter = (atomic_t *) filp->private_data;
  // Definimos las variables globales
    int v, len;
  char tmp[TMPSIZE];
    // Leemos atomico <- Es un variable atomica
  v = atomic_read(counter);
  if (*offset > 0)
    v -= 1; // el valor devuelto cuando el desplazamiento era cero
  else
    atomic_inc(counter);
  len = snprintf(tmp, TMPSIZE, "%d\n", v);
  if (*offset > len)
    return 0;
  if (count > len - *offset)
    count = len - *offset;

  // Copiamos e incrementalos
  if (copy_to_user(buf, tmp + *offset, count))
    return -EFAULT;
  *offset += count;
  return count;
}
```

5.4 Función para escribir un archivo

```c
/*
 * Write a file.
 */
static ssize_t lfs_write_file(struct file *filp, const char *buf,
    size_t count, loff_t *offset)
{
  // Extramos el contador del archivo
  atomic_t *counter = (atomic_t *) filp->private_data;
  char tmp[TMPSIZE];

  // Solo se puede escribir desde el inicio
  if (*offset != 0)
    return -EINVAL;
  // Leer el valor por el usuario
  if (count >= TMPSIZE)
    return -EINVAL;
  memset(tmp, 0, TMPSIZE);
  if (copy_from_user(tmp, buf, count))
    return -EFAULT;
  // Guarar la informacion en el contador
  atomic_set(counter, simple_strtol(tmp, NULL, 10));
  return count;
}
```

5.5 Operaciones de un archivo

```c
static struct file_operations lfs_file_ops = {
  .open = lfs_open,
  .read   = lfs_read_file,
  .write  = lfs_write_file,
};
```

5.6 Función general para crear un archivo

```c
static int simple_create(struct inode *inode, struct dentry *dentry, umode_t mode
    , bool excl)
{
    /* Simplemente realizamos el llamado a la funcion simple_mknod para
       la creacion del nodo */
  return simple_mknod(inode, dentry, mode, 0);
}
```

5.7 Función general para crear un directorio

```c
static int simple_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode)
{
  dentry = lfs_create_dir(dentry->d_sb, dentry->d_parent, dentry->d_name.name);
  return 0;
}
```

5.8 Funcion para crear un nodo

```c
static int simple_mknod(struct inode *dir, struct dentry *dentry, umode_t mode,
    dev_t dev)
{
    /* Ya que la funcion proporcionada por lwnfs para la creacion
       de archivos se requirere el contador. Para este caso usamos la funcion
    atomic_inc() vista en el curso para realizar el incremento cada vez que se
    cree un nuevo nodo */
```

```
    // Incrementamos el contador
  atomic_inc(&counter);
    // Realizamos el llamado a la funcion create_file
    dentry = lfs_create_file(dentry->d_sb, dentry->d_parent, dentry->d_name.name,
     &counter);
  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**.

```
struct inode_operations lwnfs_dir_inode_operations = {
    // Definimos la funcion de creacion de archivos
  .create = simple_create,
  .lookup = simple_lookup,
  .link = simple_link,
  .unlink = simple_unlink,
    // Definimos la funcion de creacion de directorios
  .mkdir = simple_mkdir,
  .rmdir = simple_rmdir,
    // Definimos la funcion de creacion de nodos
  .mknod = simple_mknod,
};
```

5.10 Funcion de inicialización del super block y del directorio raiz

```
static void lfs_create_files (struct super_block *sb, struct dentry *root)
{
  struct dentry *subdir;
  // Setea como contador a 0
  atomic_set(&counter, 0);
  lfs_create_file(sb, root, "counter", &counter);
  // Setea como subcontador a 0
  atomic_set(&subcounter, 0);
  subdir = lfs_create_dir(sb, root, "subdir");
  if (subdir)
    lfs_create_file(sb, subdir, "subcounter", &subcounter);
}
```

5.11 Funciones generales de inicializacion

Se declara las operaciones para el super_block

```
static struct super_operations lfs_s_ops = {
  .statfs   = simple_statfs,
  .drop_inode = generic_delete_inode,
};
```

5.12 Registro del file system

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

```
static int lfs_fill_super (struct super_block *sb, void *data, int silent)
{
  struct inode *root;
  struct dentry *root_dentry;
/*
 * Basic parameters.
 */
```

```
    /* Cambio por la version de Linux Kernel */
  sb->s_blocksize = PAGE_SIZE;
  sb->s_blocksize_bits = PAGE_SHIFT;
  sb->s_magic = LFS_MAGIC;
  sb->s_op = &lfs_s_ops;
/*
 * We need to conjure up an inode to represent the root directory
 * of this filesystem.  Its operations all come from libfs, so we
 * don't have to mess with actually *doing* things inside this
 * directory.
 */
  root = lfs_make_inode (sb, S_IFDIR | 0755);
  if (! root)
    goto out;
    /* Definimos las operaciones de lwnfs para el directorio raiz */
  root->i_op = &lwnfs_dir_inode_operations;
  root->i_fop = &simple_dir_operations;
/*
 * Get a dentry to represent the directory in core.
 */
    /* Cambio por la version de Linux Kernel */
  root_dentry = d_make_root(root);
  if (! root_dentry)
    goto out_iput;
  sb->s_root = root_dentry;
/*
 * Make up the files which will be in this filesystem, and we're done.
 */
  lfs_create_files (sb, root_dentry);
  return 0;

  out_iput:
  iput(root);
  out:
  return -ENOMEM;
}
```

5.13 Funciones finales

Registro del file system, dar de baja el filesystem.

```
/*
 * Get things set up.
 */
static int __init lfs_init(void)
{
  return register_filesystem(&lfs_type);
}

static void __exit lfs_exit(void)
{
  unregister_filesystem(&lfs_type);
}

module_init(lfs_init);
module_exit(lfs_exit);
```

6 Explicar Linux VFS

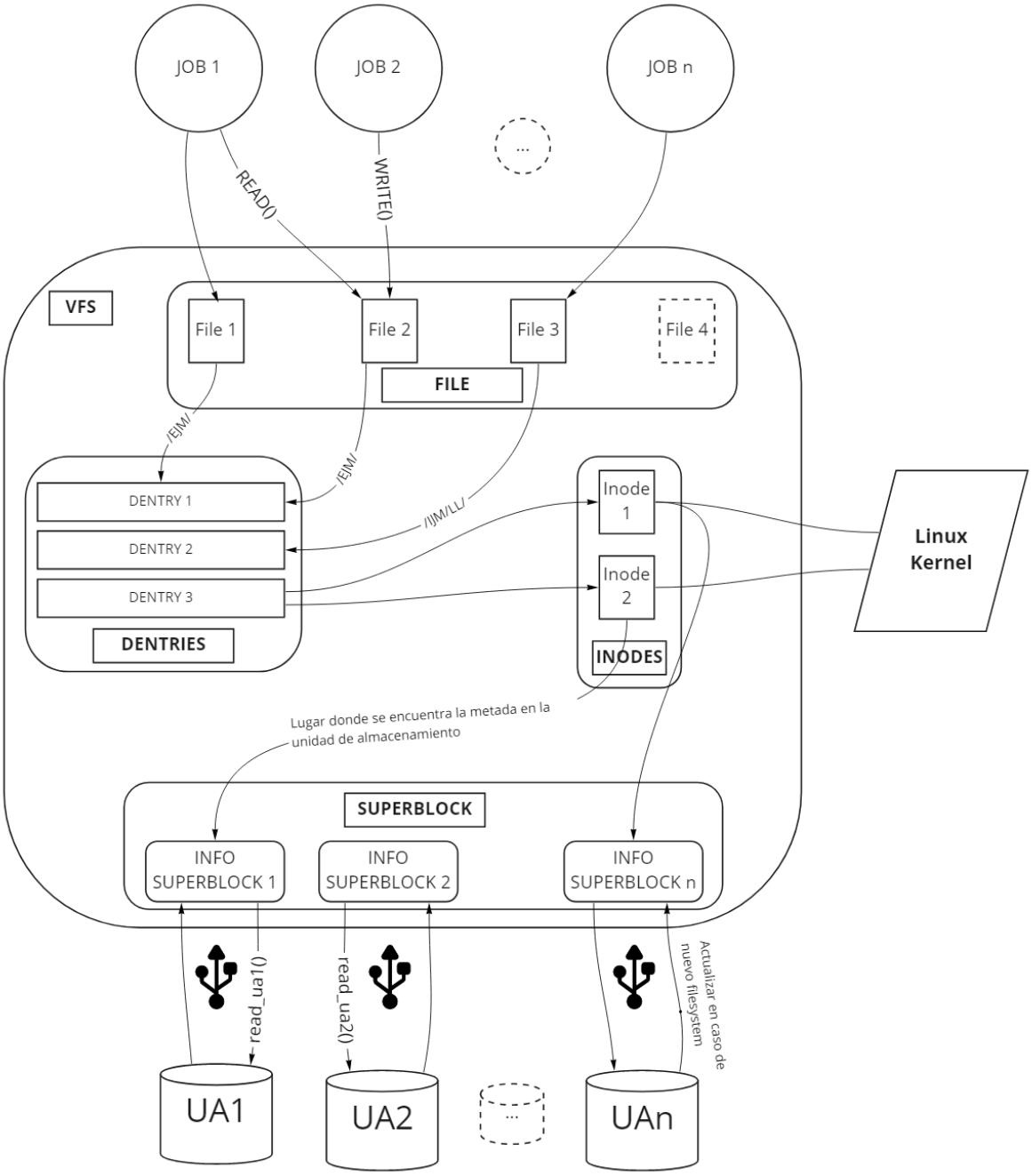
El sistema-de-archivos-virtual de Linux consiste en un ambiente abstracto virtual que permite que los programas puedan manipular cualquier sistema de archivos, y que aquellos sistemas de archivos puedan realizar operaciones entre sí. Aquello mencionado sin la necesidad de requerir algún cambio al código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

Cuando se realiza la llamada a un syscall refereido 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 funamentales (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 apunto 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 Developmente, Robert Love**.

References

[1] init_module(2): load kernel module - Linux man page. Retrieved from 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.

[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.