LECCIÓN: RAMDISK

LECCIÓN: RAMDISK

Ramdisk 1. Introducción

Cuando una aplicación necesita acceder con mucha frecuencia a datos en memoria las operaciones de entrada/salida son mucho más lentas en discos rígidos que con memoria volátil (RAM). Si sabemos esto de antemano es interesante contar con un mecanismo que permita volcar de alguna manera la información del disco duro a la memoria principal y obtener así accesos más rápidos. Con esta idea surge el concepto de RAM Disk, una reserva de espacio en memoria RAM que funciona como un dispositivo más del sistema, una unidad virtual.

Un dispositivo de tipo RAM disk se configura y utiliza de cara al sistema operativo como una partición más, por lo que le son aplicables desde línea de comandos las órdenes de montaje de particiones (*mount, umount, etc.*). El inconveniente que tiene es que al estar alojada en memoria principal los datos no permanecen indefinidamente, sino que desaparecen al apagar o reiniciar el sistema. Por ello es muy importante guardar los datos de nuevo en el disco duro siempre que se termine de usar una aplicación que haga uso de un RAM disk.

Linux maneja diferentes tipos de dispositivos de entrada/salida, así como distintos métodos de acceso a los mismos. En este sentido podemos diferenciar dispositivos de acceso en modo bloque o acceso en modo carácter. Los RAM disk se caracterizan por ser dispositivos de acceso en modo bloque, realizando las transacciones a través del Buffer Caché[1] que gestiona el controlador del módulo (RAM disk manager). La implementación se encuentra en drivers/block/brd.c.

Es importante notar que para que el núcleo soporte la creación de estos dispositivos debe estar compilado con la opción CONFIG_BLK_DEV_RAM, así como otros parámetros que mencionaremos más adelante. En las siguientes secciones además veremos como se gestionan los métodos de escritura y lectura de datos, así como las estructuras utilizadas.

El RAM Disk soporta hasta 16 discos por defecto, aunque este parámetro puede ser cambiado hasta un número ilimitado (asumiendo el riesgo de que no haya suficiente recursos hardware). Para esto debe modificarse el valor del parámetro BLK_DEV_RAM_COUNT en el menú de configuración de los dispositivos de bloque, y una vez hecho esto, recompilar el núcleo.

Para usar el soporta RAM Disk con el sistema, se debe ejecutar el comando ‘./MAKEDEV ram’ desde el directorio /dev. El número mayor de todos los discos RAM es 1, empezando con el número menor igual a 0 para /dev/ram0, etc. Los núcleos actuales usan este último dispositivo mencionado como initrd.

[1] El Buffer Caché es uno de los dos mecanismos Disk Cache de Linux destinado a agilizar las transacciones en memoria mediante buffers, aprovechando la localidad temporal de los datos. Cada buffer se asocia a un bloque del disco.

La nueva implementación de RAM Disk también tiene la habilidad de cargar imágenes comprimidas de discos RAM, permitiendo así la introducción de más programas en una instalación promedia o en un floppy de arranque.

Parámetros de configuración del núcleo

Aparte de la opción CONFIG_BLK_DEV_RAM hay otras variables que deben ser inicializadas para el correcto funcionamiento del módulo de gestión de los RAM disk (RAM disk manager). Todas ellas tienen que ver con el comportamiento que debe asumirse durante la compilación del núcleo. Por defecto vienen configuradas con unos valores, que si bien pueden cambiarse hay que tener cuidado para no sobrepasar las limitaciones del hardware o los requisitos del cargador de arranque.

ramdisk_size = N

Especifica al controlador de RAM Disk que debe configurar los discos RAM del tamaño declarado como N en kbytes. Por defecto este valor es 4096 (4 MB).

ramdisk_blocksize = N

Especifica al controlador de RAM Disk cuántos bytes deben usarse por bloque, es decir, el tamaño de bloque del dispositivo RAM Disk. Por defecto, este parámetro tiene valor 1024 (1MB).

Ramdisk 2. Estructuras utilizadas

En la implementación del RAM disk manager se utilizan distintas estructuras para gestionar todas las operaciones relacionadas con la creación y manipulación de los RAM disk. Algunas son importadas de otros módulos, como las que permiten identificar dispositivos (struct device,...), y otras se definen localmente. Las mencionadas aquí son estas últimas. linux/drivers/block/brd.c

Estructura que representa al dispositivo RAM Disk

```
struct brd_device {
 36       int             brd_number;                     //número de dispositivo
 37       int             brd_refcnt;
 38       loff_t          brd_offset;                     //zona de memoria de comienzo
 39       loff_t          brd_sizelimit;                  //tamaño de memoria del dispositivo
 40       unsigned        brd_blocksize;                  //tamaño de bloque del dispositivo
 41
 42       struct request_queue    *brd_queue;             //cola de operaciones
 43       struct gendisk          *brd_disk;              //disco asociado
 44       struct list_head        brd_list;               //lista doblemente encadenada usada en el loop
 45
 46       /*
 47        * Backing store of pages and lock to protect it. This is the contents
 48        * of the block device.
 49        */
 50       spinlock_t              brd_lock;               //cerrojo de acceso al árbol de páginas
 51       struct radix_tree_root  brd_pages;              //raíz del árbol de páginas
 52};
```

Estructura encargada de actuar como interfaz para llamadas al núcleo

```
static struct block_device_operations brd_fops = {
 377       .owner =                THIS_MODULE,
 378       .ioctl =                brd_ioctl,
 379#ifdef CONFIG_BLK_DEV_XIP
 380       .direct_access =        brd_direct_access,
 381#endif
 382};
```

Este módulo hace uso de la estructura radix_tree_root, cuya especificación se encuentra en el fichero /include/linux/radix-tree.h

```
 61struct radix_tree_root {
 62       unsigned int            height;
 63       gfp_t                   gfp_mask;
 64       struct radix_tree_node  *rnode;
 65};
```

Es una estructura de tipo árbol, donde cada nodo representa una página. Algunas de las operaciones utilizadas son:

```
/*inserta un elemento en el árbol de páginas, en la posición indicada*/
 155int radix_tree_insert(struct radix_tree_root *, unsigned long, void *);

/*busca un elemento en el árbol de páginas*/
 156void *radix_tree_lookup(struct radix_tree_root *, unsigned long);

/*elimina el elemento indicado del árbol*/
 158void *radix_tree_delete(struct radix_tree_root *, unsigned long);

/*busca y devuelve todo los elementos del árbol, desde el first_index hasta un máximo de max_items elementos*/
 160radix_tree_gang_lookup(struct radix_tree_root *root, void **results,
 161                        unsigned long first_index, unsigned int max_items);

/*asegura que haya espacio en el árbol para una nueva página y lo reserva*/
 164int radix_tree_preload(gfp_t gfp_mask);

/*terminar la reserva del espacio y devolver al sistema a su estado habitual*/
 178static inline void radix_tree_preload_end(void)
 179{
 180        preempt_enable();
 181}
```

Hace uso de las estructuras bio y gendisk, ya vistas en otros capítulos.

Ramdisk 3. Funciones

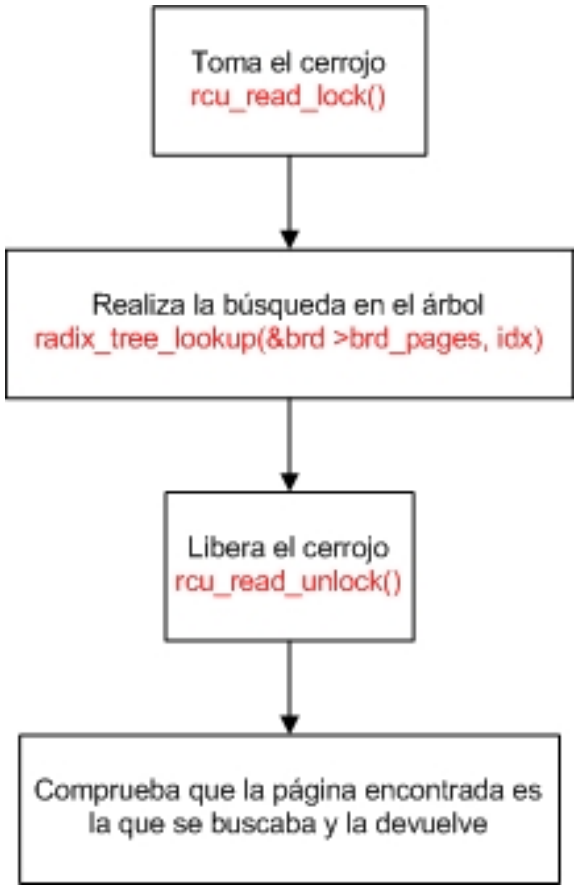
A continuación se detallan cada una de las funciones del módulo linux/drivers/block/brd.c que contribuyen en el funcionamiento del manejador que estamos tratando:

brd_lookup_page

```
/*
  55 * Look up and return a brd's page for a given sector.
  56 */
  57static struct page *brd_lookup_page(struct brd_device *brd, sector_t sector)
  58{
  59        pgoff_t idx;
  60        struct page *page;
  61
  62        /*
  63         * The page lifetime is protected by the fact that we have opened the
  64         * device node -- brd pages will never be deleted under us, so we
  65         * don't need any further locking or refcounting.
  66         *
  67         * This is strictly true for the radix-tree nodes as well (ie. we
  68         * don't actually need the rcu_read_lock()), however that is not a
  69         * documented feature of the radix-tree API so it is better to be
  70         * safe here (we don't have total exclusion from radix tree updates
```

```
71        * here, only deletes).
72        */
73       rcu_read_lock();
74       idx = sector >> PAGE_SECTORS_SHIFT; /* sector to page index */
75       page = radix_tree_lookup(&brd->brd_pages, idx);
76       rcu_read_unlock();
77
78       BUG_ON(page && page->index != idx);
79
80       return page;
81}
```

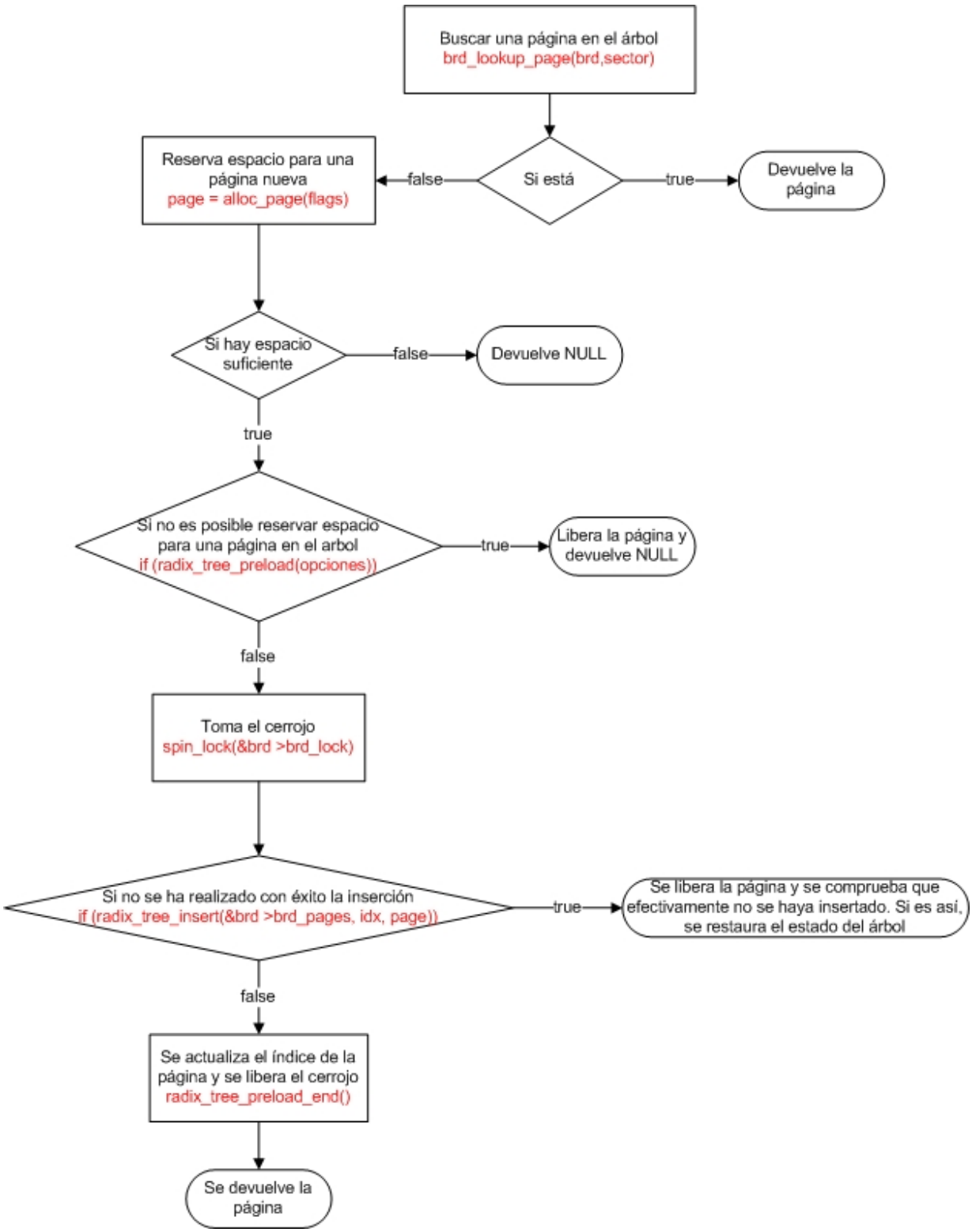
Esta función busca una página determinada por el parámetro sector que se le especifica. Si la encuentra, devuelve un puntero a la misma, y NULL en caso contrario. A continuación se muestra su diagrama de flujo:

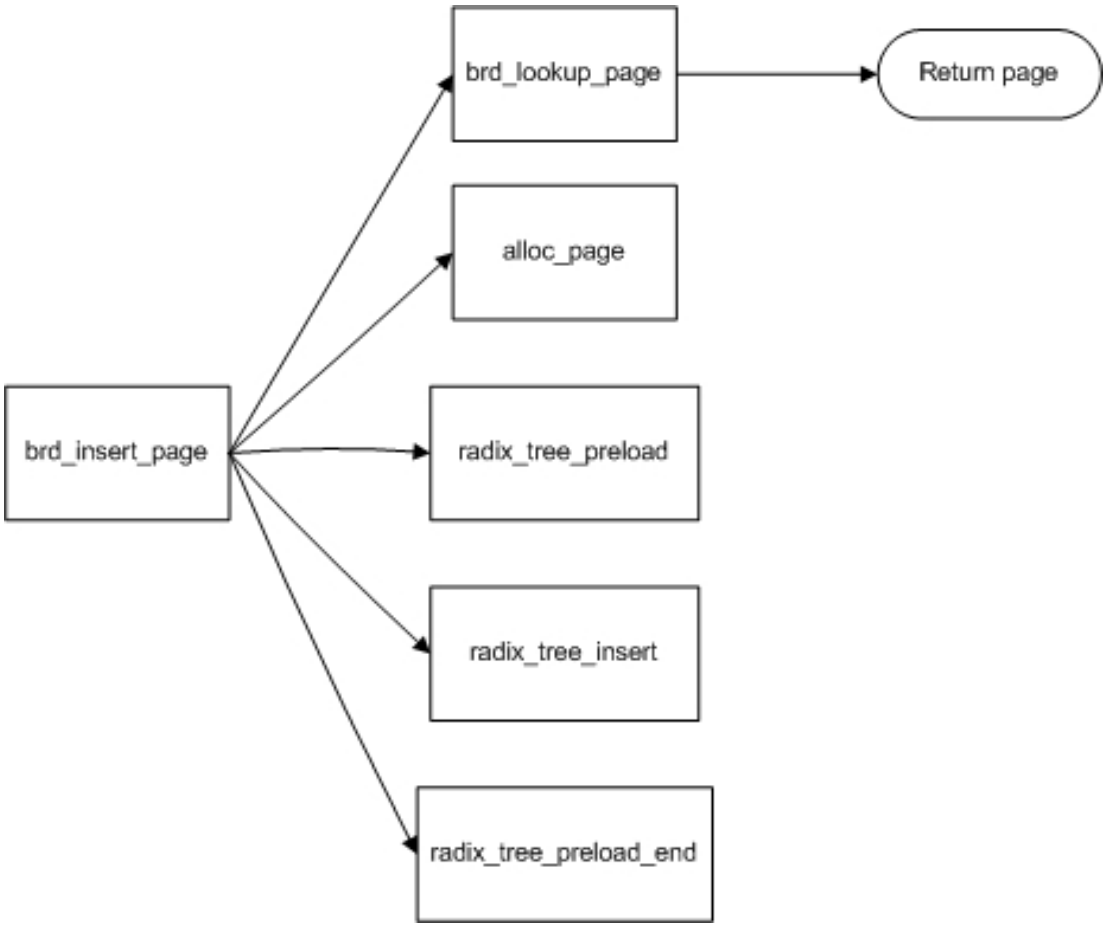


brd_insert_page:

```
/*
 84 * Look up and return a brd's page for a given sector.
 85 * If one does not exist, allocate an empty page, and insert that. Then
 86 * return it.
 87 */
 88static struct page *brd_insert_page(struct brd_device *brd, sector_t sector)
 89{
 90        pgoff_t idx;
 91        struct page *page;
 92        gfp_t gfp_flags;
 93
 94        page = brd_lookup_page(brd, sector);
 95        if (page)
 96                return page;
 97
 98        /*
 99         * Must use NOIO because we don't want to recurse back into the
100         * block or filesystem layers from page reclaim.
101         *
102         * Cannot support XIP and highmem, because our ->direct_access
103         * routine for XIP must return memory that is always addressable.
104         * If XIP was reworked to use pfns and kmap throughout, this
105         * restriction might be able to be lifted.
106         */
107        gfp_flags = GFP_NOIO | __GFP_ZERO;
108#ifndef CONFIG_BLK_DEV_XIP
109        gfp_flags |= __GFP_HIGHMEM;
110#endif
111        page = alloc_page(GFP_NOIO | __GFP_HIGHMEM | __GFP_ZERO);
112        if (!page)
113                return NULL;
114
115        if (radix_tree_preload(GFP_NOIO)) {
116                __free_page(page);
117                return NULL;
118        }
119
120        spin_lock(&brd->brd_lock);
121        idx = sector >> PAGE_SECTORS_SHIFT;
122        if (radix_tree_insert(&brd->brd_pages, idx, page)) {
123                __free_page(page);
124                page = radix_tree_lookup(&brd->brd_pages, idx);
125                BUG_ON(!page);
126                BUG_ON(page->index != idx);
127        } else
128                page->index = idx;
129        spin_unlock(&brd->brd_lock);
130
131        radix_tree_preload_end();
132
133        return page;
134}
```

Esta función busca una página desde la posición indicada por el parámetro sector. Si existe devuelve un puntero a la misma, y si no, crea una vacía y la inserta en el árbol, en caso de que no haya ningún problema. Seguidamente se muestra los diferentes pasos del procedimiento:



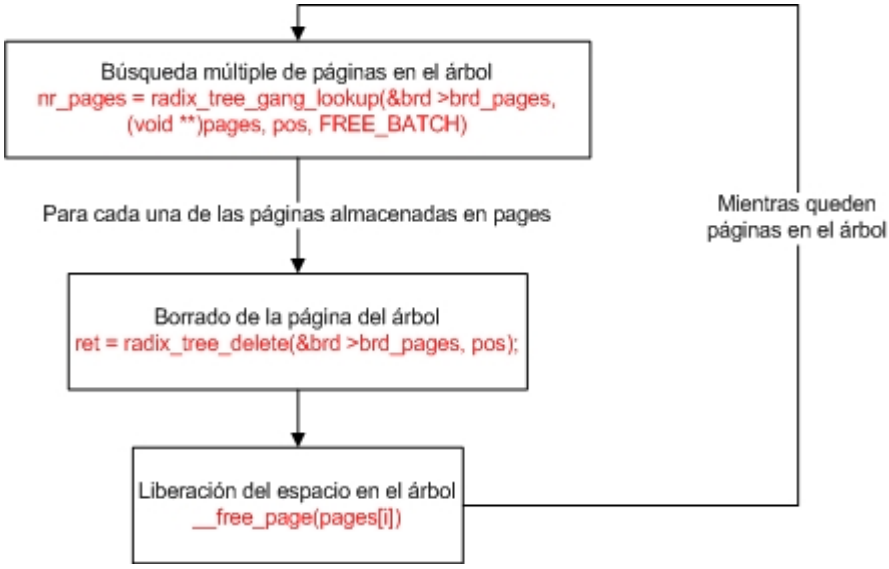


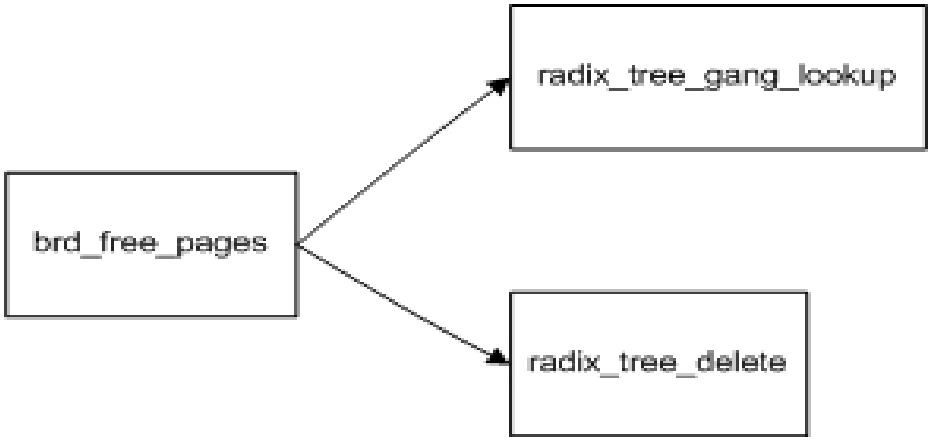
brd_free_pages

```
/*
 137 * Free all backing store pages and radix tree. This must only be called when
 138 * there are no other users of the device.
 139 */
 140#define FREE_BATCH 16
 141static void brd_free_pages(struct brd_device *brd)
 142{
 143        unsigned long pos = 0;
 144        struct page *pages[FREE_BATCH];
 145        int nr_pages;
 146
 147        do {
 148                int i;
 149
 150                nr_pages = radix_tree_gang_lookup(&brd->brd_pages,
 151                                (void **)pages, pos, FREE_BATCH);
 152
 153                for (i = 0; i < nr_pages; i++) {
 154                        void *ret;
 155
 156                        BUG_ON(pages[i]->index < pos);
 157                        pos = pages[i]->index;
 158                        ret = radix_tree_delete(&brd->brd_pages, pos);
 159                        BUG_ON(!ret || ret != pages[i]);
 160                        __free_page(pages[i]);
 161                }
```

```
162
163             pos++;
164
165             /*
166              * This assumes radix_tree_gang_lookup always returns as
167              * many pages as possible. If the radix-tree code changes,
168              * so will this have to.
169              */
170     } while (nr_pages == FREE_BATCH);
171}
```

Va liberando las páginas del árbol una a una, y finalmente pone a nulo el puntero a la raíz del árbol. En otras palabras, elimina las páginas y libera el espacio que tenían reservado en memoria. Veámos como esto es llevado a cabo de una manera esquemática:





copy_to_brd_setup

```
/*
 174 * copy_to_brd_setup must be called before copy_to_brd. It may sleep.
 175 */
 176static int copy_to_brd_setup(struct brd_device *brd, sector_t sector, size_t n)
 177{
 178        unsigned int offset = (sector & (PAGE_SECTORS-1)) << SECTOR_SHIFT;
 179        size_t copy;
 180
 181        copy = min_t(size_t, n, PAGE_SIZE - offset);
 182        if (!brd_insert_page(brd, sector))
 183                return -ENOMEM;
 184        if (copy < n) {
 185                sector += copy >> SECTOR_SHIFT;
 186                if (!brd_insert_page(brd, sector))
 187                        return -ENOMEM;
 188        }
 189        return 0;
 190}
```

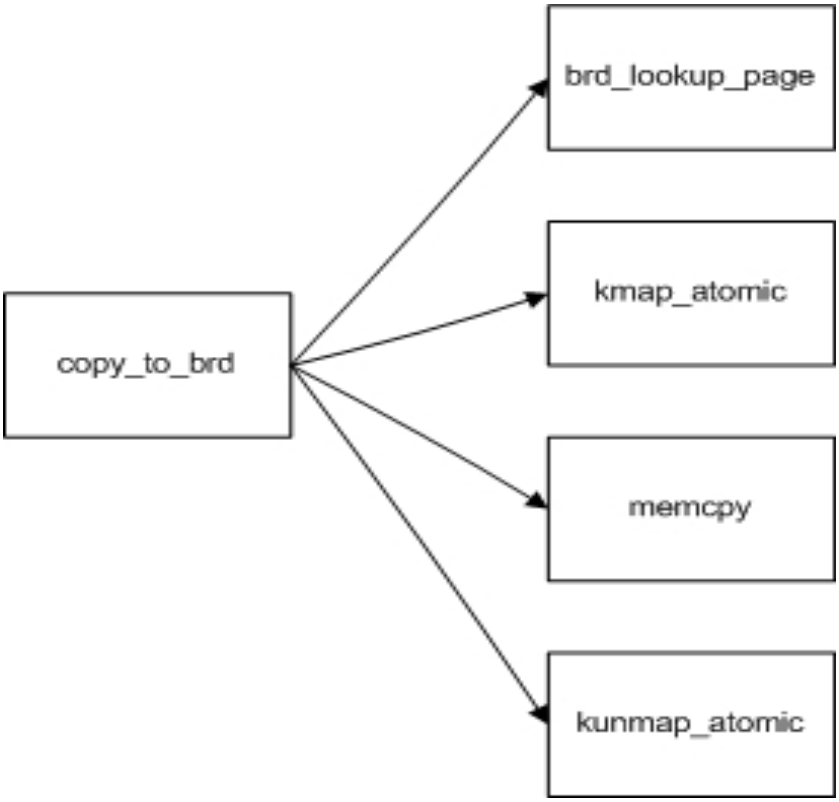
Esta función prepara el dispositivo para realizar una copia, es decir, reserva el espacio, introduciendo nuevas páginas en el árbol, de acuerdo con el tamaño especificado por el parámetro n. La reserva comienza en la dirección de memoria especificada por el parámetro sector.

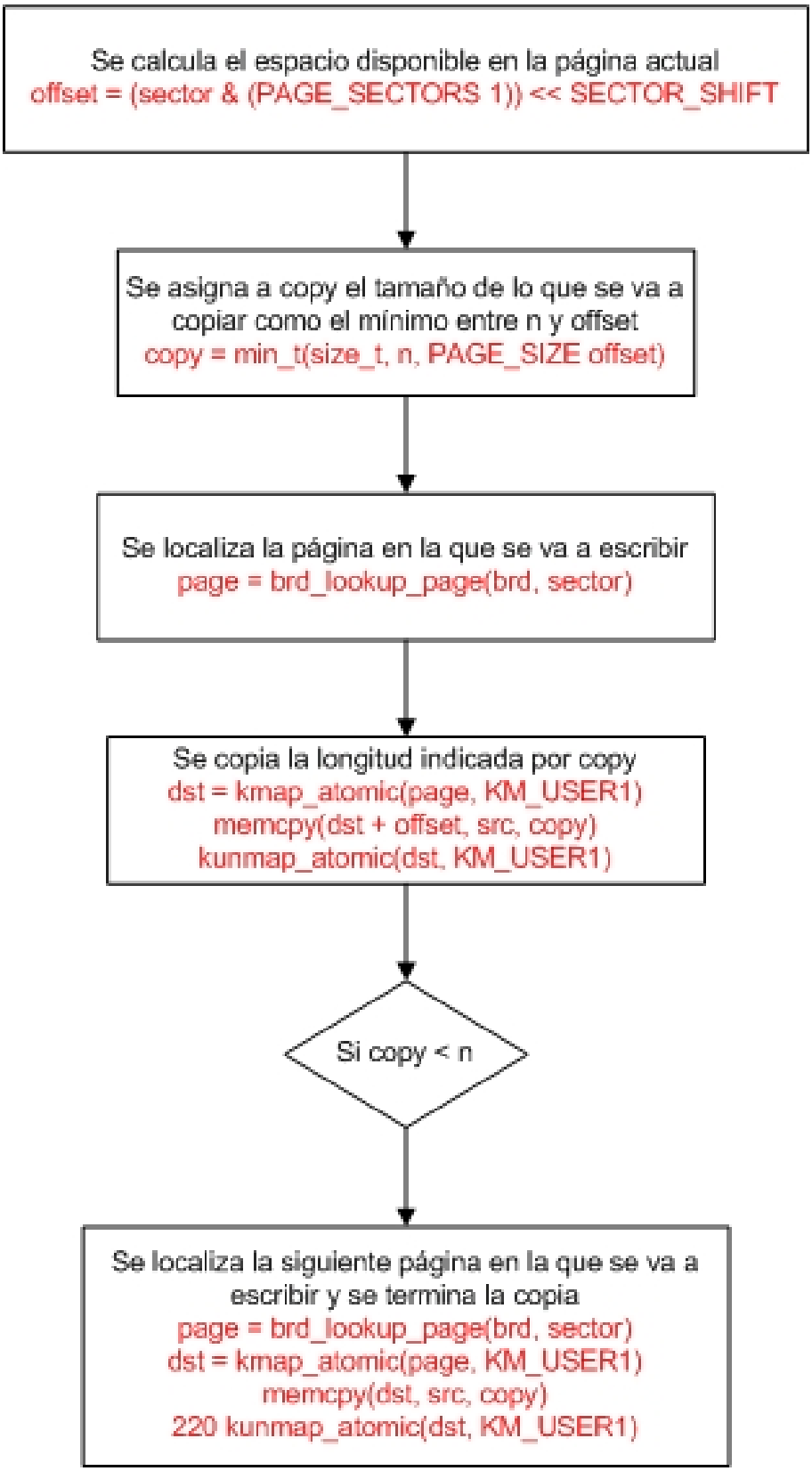
copy_to_brd

```
/*
 193 * Copy n bytes from src to the brd starting at sector. Does not sleep.
 194 */
 195static void copy_to_brd(struct brd_device *brd, const void *src,
 196                        sector_t sector, size_t n)
 197{
 198        struct page *page;
 199        void *dst;
 200        unsigned int offset = (sector & (PAGE_SECTORS-1)) << SECTOR_SHIFT;
 201        size_t copy;
 202
 203        copy = min_t(size_t, n, PAGE_SIZE - offset);
 204        page = brd_lookup_page(brd, sector);
 205        BUG_ON(!page);
 206
 207        dst = kmap_atomic(page, KM_USER1);
 208        memcpy(dst + offset, src, copy);
 209        kunmap_atomic(dst, KM_USER1);
 210
 211        if (copy < n) {
 212                src += copy;
 213                sector += copy >> SECTOR_SHIFT;
 214                copy = n - copy;
 215                page = brd_lookup_page(brd, sector);
```

```
216             BUG_ON(!page);
217
218             dst = kmap_atomic(page, KM_USER1);
219             memcpy(dst, src, copy);
220             kunmap_atomic(dst, KM_USER1);
221     }}
```

Esta función se encarga de copiar n bytes desde src en el dispositivo, empezando desde donde indica el parámetro sector. Es el proceso análogo a la escritura en el dispositivo. Véase el diagrama de flujo:

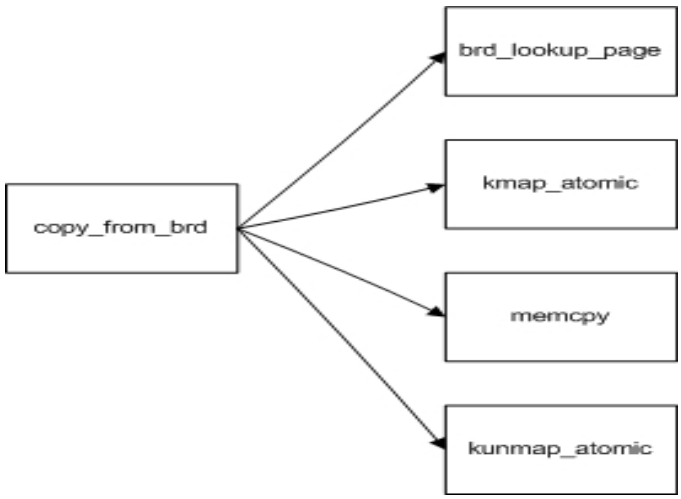


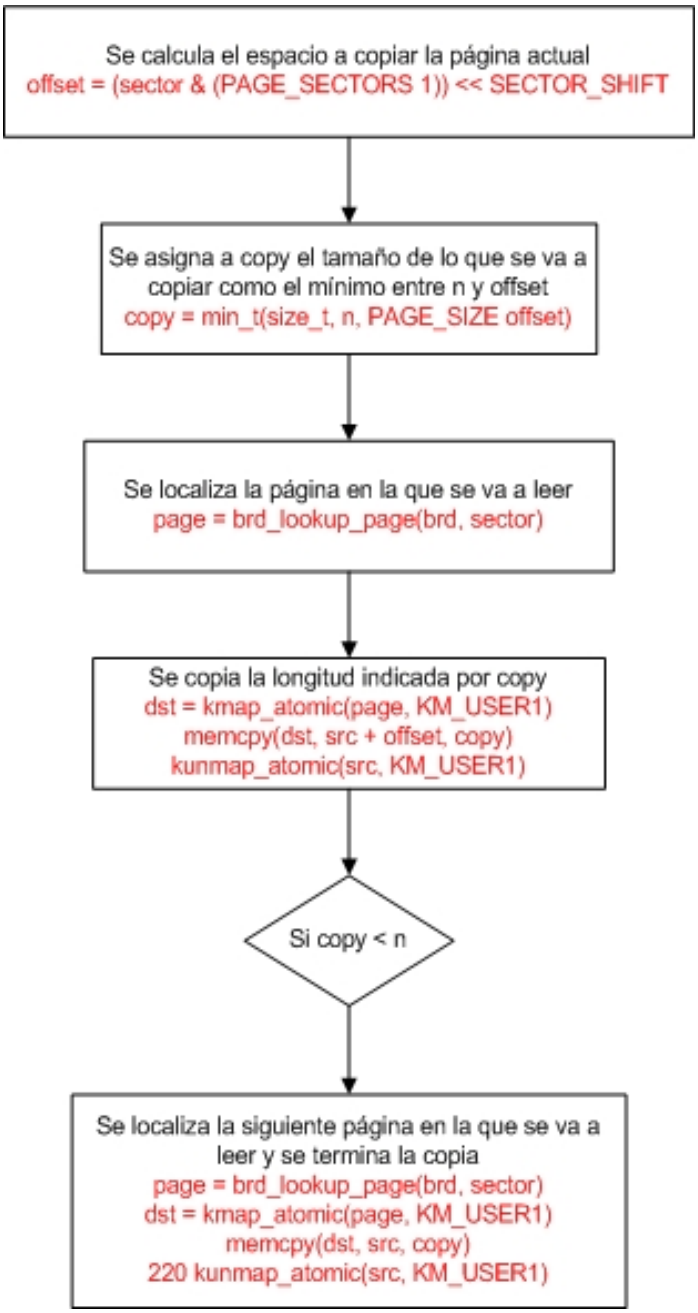

Se calcula el espacio disponible en la página actual
offset = (sector & (PAGE_SECTORS 1)) << SECTOR_SHIFT
Se asigna a copy el tamaño de lo que se va a copiar como el mínimo entre n y offset
copy = min_t(size_t, n, PAGE_SIZE offset)
Se localiza la página en la que se va a escribir
page = brd_lookup_page(brd, sector)
Se copia la longitud indicada por copy
dst = kmap_atomic(page, KM_USER1)
memcpy(dst + offset, src, copy)
kunmap_atomic(dst, KM_USER1)
Si copy < n
Se localiza la siguiente página en la que se va a escribir y se termina la copia
page = brd_lookup_page(brd, sector)
dst = kmap_atomic(page, KM_USER1)
memcpy(dst, src, copy)
220 kunmap_atomic(dst, KM_USER1)

copy_from_brd

```
/*
 225 * Copy n bytes to dst from the brd starting at sector. Does not sleep.
 226 */
 227static void copy_from_brd(void *dst, struct brd_device *brd,
 228                         sector_t sector, size_t n)
 229{
 230        struct page *page;
 231        void *src;
 232        unsigned int offset = (sector & (PAGE_SECTORS-1)) << SECTOR_SHIFT;
 233        size_t copy;
 234
 235        copy = min_t(size_t, n, PAGE_SIZE - offset);
 236        page = brd_lookup_page(brd, sector);
 237        if (page) {
 238                src = kmap_atomic(page, KM_USER1);
 239                memcpy(dst, src + offset, copy);
 240                kunmap_atomic(src, KM_USER1);
 241        } else
 242                memset(dst, 0, copy);
 243
 244        if (copy < n) {
 245                dst += copy;
 246                sector += copy >> SECTOR_SHIFT;
 247                copy = n - copy;
 248                page = brd_lookup_page(brd, sector);
 249                if (page) {
 250                        src = kmap_atomic(page, KM_USER1);
 251                        memcpy(dst, src, copy);
 252                        kunmap_atomic(src, KM_USER1);
 253                } else
 254                        memset(dst, 0, copy);
 255        }
 256}
```

Esta función copia n bytes al destino (dst) desde el dispositivo brd empezando en la dirección de memoria indicada por el parámetro sector. Es la función que implementa la lectura del dispositivo. A continuación se muestra su diagrama de flujo, muy similar al de la escritura:

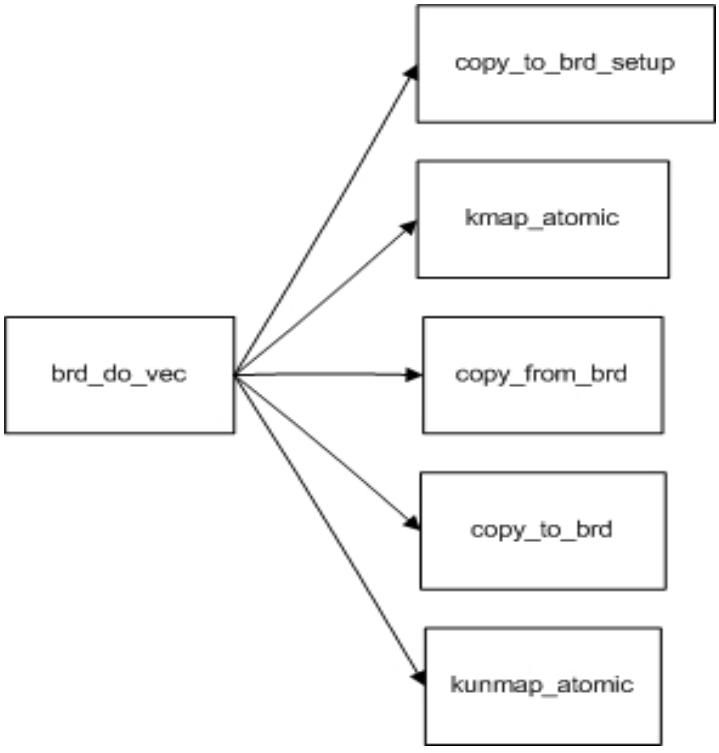


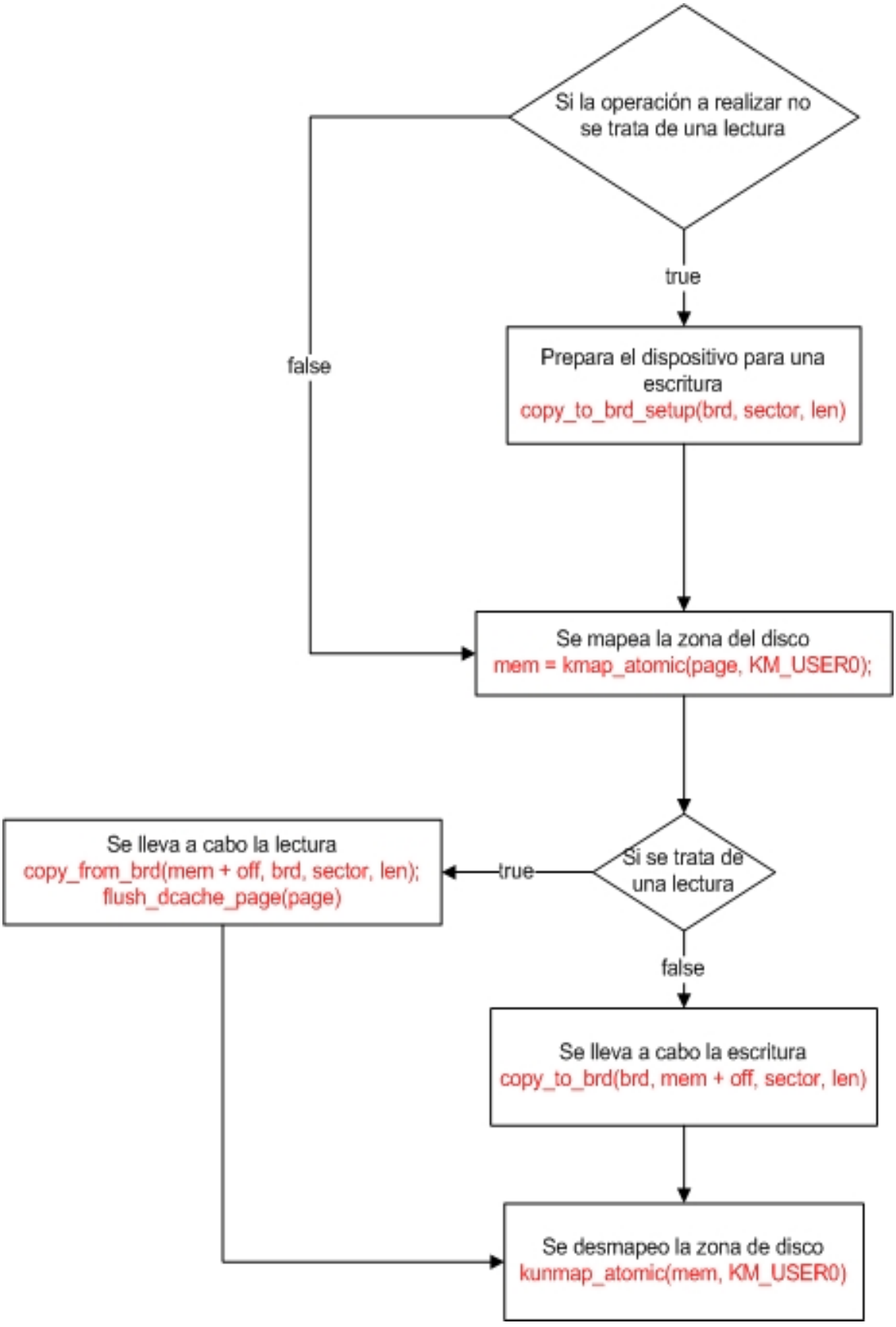

Se calcula el espacio a copiar la página actual
offset = (sector & (PAGE_SECTORS 1)) << SECTOR_SHIFT
Se asigna a copy el tamaño de lo que se va a copiar como el mínimo entre n y offset
copy = min_t(size_t, n, PAGE_SIZE offset)
Se localiza la página en la que se va a leer
page = brd_lookup_page(brd, sector)
Se copia la longitud indicada por copy
dst = kmap_atomic(page, KM_USER1)
memcpy(dst, src + offset, copy)
kunmap_atomic(src, KM_USER1)
Si copy < n
Se localiza la siguiente página en la que se va a leer y se termina la copia
page = brd_lookup_page(brd, sector)
dst = kmap_atomic(page, KM_USER1)
memcpy(dst, src, copy)
220 kunmap_atomic(src, KM_USER1)

brd_do_bvec:

```
/*
 259 * Process a single bvec of a bio.
 260 */
 261static int brd_do_bvec(struct brd_device *brd, struct page *page,
 262                        unsigned int len, unsigned int off, int rw,
 263                        sector_t sector)
 264{
 265        void *mem;
 266        int err = 0;
 267
 268        if (rw != READ) {
 269                err = copy_to_brd_setup(brd, sector, len);
 270                if (err)
 271                        goto out;
 272        }
 273
 274        mem = kmap_atomic(page, KM_USER0);
 275        if (rw == READ) {
 276                copy_from_brd(mem + off, brd, sector, len);
 277                flush_dcache_page(page);
 278        } else
 279                copy_to_brd(brd, mem + off, sector, len);
 280        kunmap_atomic(mem, KM_USER0);
 281
 282out:
 283        return err;
 284}
```

Esta función ejecuta una operación de entrada/salida, que será una lectura o escritura en el dispositivo brd en función de lo que indique el parámetro entero rw y a partir de la posición de memoria indicada por el parámetro sector. Ésta es la encargada de llamar a las dos funciones anteriores. Su diagrama de fujo es el que sigue:

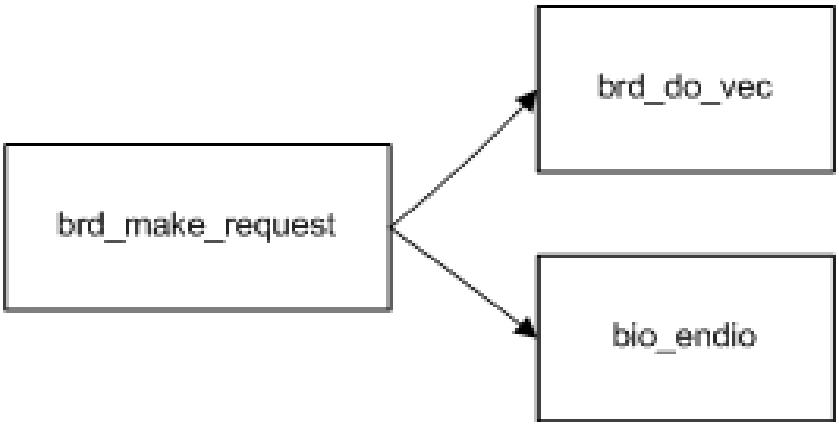


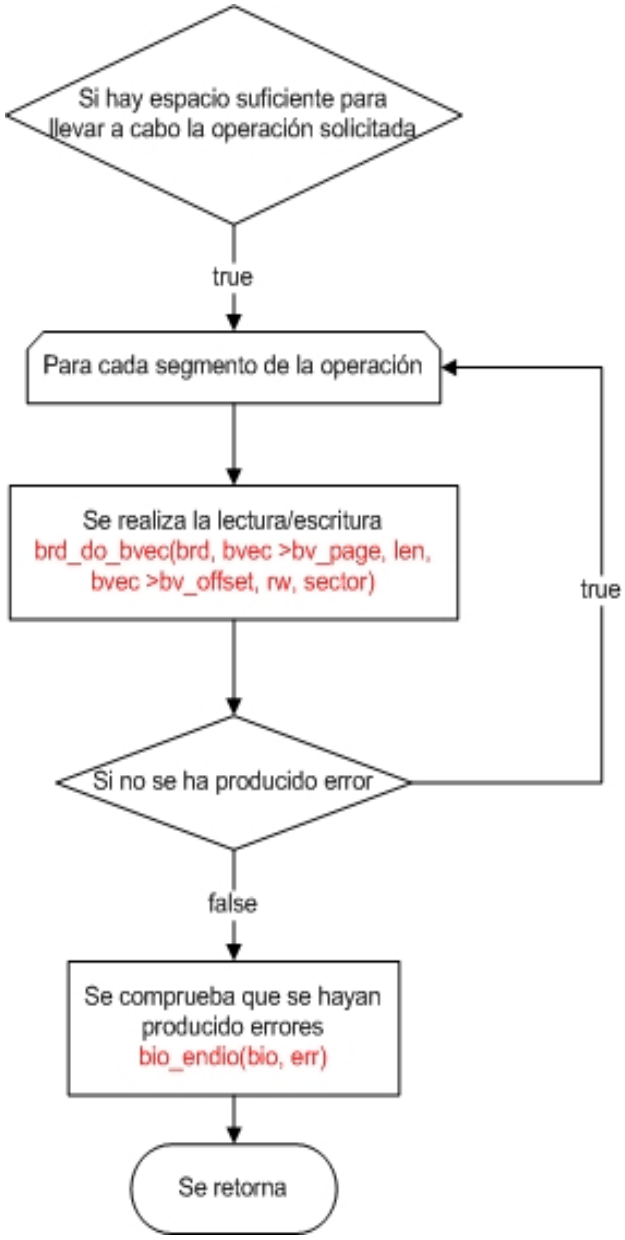

Si la operación a realizar no se trata de una lectura
true
false
Prepara el dispositivo para una escritura
copy_to_brd_setup(brd, sector, len)
Se mapea la zona del disco
mem = kmap_atomic(page, KM_USER0);
Si se trata de una lectura
true
Se lleva a cabo la lectura
copy_from_brd(mem + off, brd, sector, len);
flush_dcache_page(page)
false
Se lleva a cabo la escritura
copy_to_brd(brd, mem + off, sector, len)
Se desmapeo la zona de disco
kunmap_atomic(mem, KM_USER0)

brd_make_request:

```
static int brd_make_request(struct request_queue *q, struct bio *bio)
 287{
 288        struct block_device *bdev = bio->bi_bdev;
 289        struct brd_device *brd = bdev->bd_disk->private_data;
 290        int rw;
 291        struct bio_vec *bvec;
 292        sector_t sector;
 293        int i;
 294        int err = -EIO;
 295
 296        sector = bio->bi_sector;
 297        if (sector + (bio->bi_size >> SECTOR_SHIFT) >
 298                                        get_capacity(bdev->bd_disk))
 299                goto out;
 300
 301        rw = bio_rw(bio);
 302        if (rw == READA)
 303                rw = READ;
 304
 305        bio_for_each_segment(bvec, bio, i) {
 306                unsigned int len = bvec->bv_len;
 307                err = brd_do_bvec(brd, bvec->bv_page, len,
 308                                        bvec->bv_offset, rw, sector);
 309                if (err)
 310                        break;
 311                sector += len >> SECTOR_SHIFT;
 312        }
 313
 314out:
 315        bio_endio(bio, err);
 316
 317        return 0;
 318}
```

Se solicita una operación de entrada/salida a través de esta función. Hace las veces de interfaz entre el sistema y el módulo que estamos tratando para la llamada a la función brd_do_vec. A continuación se muestra el proceso que sigue:

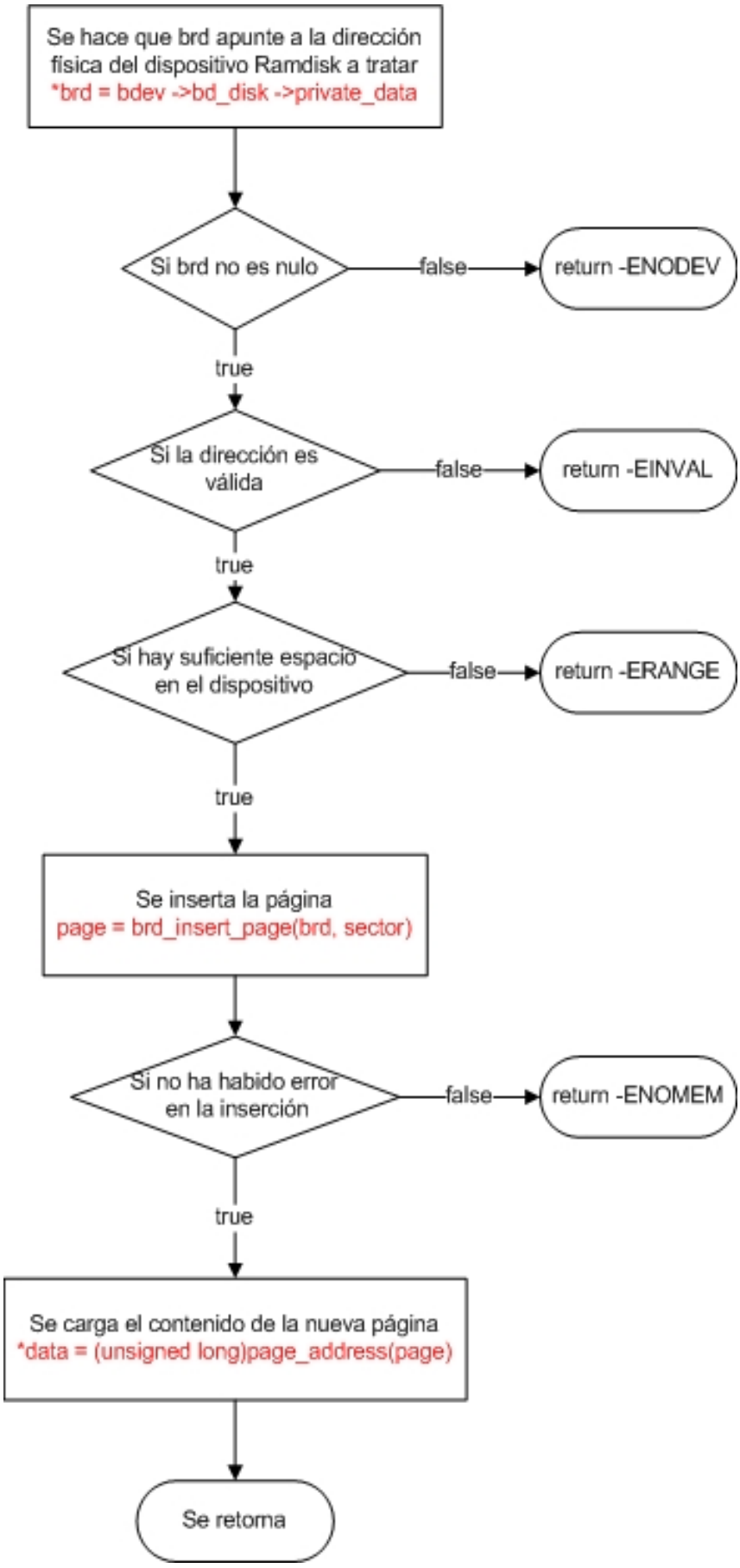

brd_make_request
brd_do_vec
bio_endio


Si hay espacio suficiente para llevar a cabo la operación solicitada
true
Para cada segmento de la operación
Se realiza la lectura/escritura
brd_do_bvec(brd, bvec >bv_page, len, bvec >bv_offset, rw, sector)
Si no se ha producido error
true
false
Se comprueba que se hayan producido errores
bio_endio(bio, err)
Se retorna

brd_direct_access:

```
#ifdef CONFIG_BLK_DEV_XIP
 321static int brd_direct_access (struct block_device *bdev, sector_t sector,
 322                        unsigned long *data)
 323{
 324        struct brd_device *brd = bdev->bd_disk->private_data;
 325        struct page *page;
 326
 327        if (!brd)
 328                return -ENODEV;
 329        if (sector & (PAGE_SECTORS-1))
 330                return -EINVAL;
 331        if (sector + PAGE_SECTORS > get_capacity(bdev->bd_disk))
 332                return -ERANGE;
 333        page = brd_insert_page(brd, sector);
 334        if (!page)
 335                return -ENOMEM;
 336        *data = (unsigned long)page_address(page);
 337
 338        return 0;
 339}
 340#endif
```

Esta función define el acceso directo si está activada la opción de configuración CONF_BLK_DEV_XIP, que permite el XIP filesystems en un dispositivo RAM. No se necesita hacer mapeo ni uso de funciones del módulo para llevar a cabo la lectura en el dispositivo. Busca las páginas en el árbol directamente. A continuación se muestra su diagrama de flujo:


Se hace que brd apunte a la dirección física del dispositivo Ramdisk a tratar
*brd = bdev ->bd_disk ->private_data
Si brd no es nulo
false
return -ENODEV
true
Si la dirección es válida
false
return -EINVAL
true
Si hay suficiente espacio en el dispositivo
false
return -ERANGE
true
Se inserta la página
page = brd_insert_page(brd, sector)
Si no ha habido error en la inserción
false
return -ENOMEM
true
Se carga el contenido de la nueva página
*data = (unsigned long)page_address(page)
Se retorna

brd_ioctl:

```
static int brd_ioctl(struct inode *inode, struct file *file,
 343                     unsigned int cmd, unsigned long arg)
 344{
 345        int error;
 346        struct block_device *bdev = inode->i_bdev;
 347        struct brd_device *brd = bdev->bd_disk->private_data;
 348
 349        if (cmd != BLKFLSBUF)
 350                return -ENOTTY;
 351
 352        /*
 353         * ram device BLKFLSBUF has special semantics, we want to actually
 354         * release and destroy the ramdisk data.
 355         */
 356        mutex_lock(&bdev->bd_mutex);
 357        error = -EBUSY;
 358        if (bdev->bd_openers <= 1) {
 359                /*
 360                 * Invalidate the cache first, so it isn't written
 361                 * back to the device.
 362                 *
 363                 * Another thread might instantiate more buffercache here,
 364                 * but there is not much we can do to close that race.
 365                 */
 366                invalidate_bh_lrus();
 367                truncate_inode_pages(bdev->bd_inode->i_mapping, 0);
 368                brd_free_pages(brd);
 369                error = 0;
 370        }
 371        mutex_unlock(&bdev->bd_mutex);
 372
 373        return error;
 374}
```

Implementa la llamada al sistema ioctl. Si el comando cmd especificado es diferente de BLKFLSBUF, devuelve un error, indicando que se trata de un comando no válido. En otro caso, libera la memoria asociada al inode y elimina el dispositivo RAM, liberando la memoria reservada con la función brd_free_pages(brd).

- **Configuración de parámetros:**

```
/*
 385 * And now the modules code and kernel interface.
 386 */
 387static int rd_nr;
 388int rd_size = CONFIG_BLK_DEV_RAM_SIZE;
 389module_param(rd_nr, int, 0);
 390MODULE_PARM_DESC(rd_nr, "Maximum number of brd devices");
 391module_param(rd_size, int, 0);
 392MODULE_PARM_DESC(rd_size, "Size of each RAM disk in kbytes.");
 393MODULE_LICENSE("GPL");
 394MODULE_ALIAS_BLOCKDEV_MAJOR(RAMDISK_MAJOR);
```

Se definen diferentes flags y opciones de configuración, que serán tomadas en cuenta por el núcleo al crear un nuevo dispositivo RAM Disk.

Por último, si en el arranque no está definida la carga modular, será necesario utilizar las siguientes funciones de inicialización del dispositivo RAM Disk.

```
 396#ifndef MODULE
 397/* Legacy boot options - nonmodular */
 398static int __init ramdisk_size(char *str)
 399{
 400        rd_size = simple_strtol(str, NULL, 0);
 401        return 1;
 402}
 403static int __init ramdisk_size2(char *str)
 404{
 405        return ramdisk_size(str);
 406}
 407__setup("ramdisk=", ramdisk_size);
 408__setup("ramdisk_size=", ramdisk_size2);
 409#endif
```

Creación y manejo desde un terminal

En los sistemas operativos basados en Linux (y en UNIX en general) que vienen con el soporte para RAM disk por defecto sólo es necesario crear una partición con el sistema de ficheros deseado y montarla en un directorio a partir del dispositivo /dev/ram0, que ya existe. Para obtener una lista de los ramdisks disponibles en el sistema, por ejemplo puede ejecutarse el comando ls -al /dev/ram*. Mientras no se asigne un sistema de ficheros o se monten los ramdisks no ocupan memoria principal. Veamos un ejemplo de como inicializar un RAM disk:

```
# Crea el directorio donde montaremos el ramdisk
mkdir -p /tmp/ramdisk0
# Crea un sistema de ficheros en el disco (lo formatea)
mkfs -t ext2 /dev/ram0
# Monta el ramdisk en el directorio elegido
mount /dev/ram0 /tmp/ramdisk0
```

Ahora ya tendríamos un disco duro virtual en memoria RAM al que podamos acceder normalmente como si fuera cualquier otro dispositivo (floppy, cdrom, usbdisk,...), y sobre el que podemos aplicar casi todos los comandos del terminal (cp, mkdir, etc.). Si no es posible la creación del ramdisk es posible que el núcleo no haya sido compilado con soporte para RAM disk, a pesar de que estos como dispositivos estén disponibles.

Un ejemplo práctico de la utilidad de los RAM disks aparece por ejemplo en las aplicaciones de red basadas en el modelo cliente-servidor. Pensemos en un servidor web que debe atender peticiones de infinidad de clientes. Si tiene que ir al disco duro para cargar una página y mandarla al destino correspondiente la latencia sería mayor que si tenemos el contenido accesible en un RAM disk. Al iniciar el servidor podríamos volcar los contenidos a un ramdisk creado:

```
tar -C /home/httpd_real -c . | tar -C /home/httpd_ram -x
```

Carga de un RAM disk desde un dispositivo

Linux permite inicializar el contenido de un disco ram un dispositivo. Esto resulta particularmente útil en el caso de un disco de arranque, con un sistema de archivos que se carga en el RAM disk creado en el arranque del sistema.

En el inicio del sistema se llama a rd_load después de rd_init, que a su vez llama a rd_load_image en caso de que el punto de montaje (directorio /) esté dentro de un disco de arranque externo (floppy, cdrom, etc.). Si queremos que siempre se cargue un determinado dispositivo en un RAM disk en el inicio del sistema el núcleo debe estar compilado con la opción CONFIG_BLK_DEV_INITRD. Si lo hacemos en vez de llamarse a rd_load se llama a initrd_load.

Cuando se inicia el núcleo se reserva una zona de memoria para los RAM disks que se hayan cargado. Para cada uno de ellos se guardan las direcciones de inicio y fin en las variables initrd_start e initrd_end respectivamente.

Función rd_load_image

1. Abre el RAM disk y el dispositivo a leer con blk_open.
2. Identifica el tipo de sistema de ficheros del dispositivo. Soporta Minix, Ext2 e imágenes comprimidas con gzip.
3. Carga el contenido, llamando a la función read asociada al RAM disk para cada bloquue del dispositivo.
4. Los buffers que se asignaron al dispositivo fuente se invalidan y se cierra el dispositivo.

Bibliografía

- http://lxr.linux.no/source/drivers**/block/brd.c**

- Programación linux 2.0.
  API de sistema y funcionamiento del núcleo.

- http://acapulco.dyndns.org/manual/src/linux26/brd_8c.htm#e72c4ade9cd62a5db3de758c85c13522
- http://lxr.linux.no/linux+v2.6.25.4/Documentation/ramdisk.txt