LECCIÓN 16: BUFFER CACHE

LECCIÓN 16: Buffer Cache

16.1 Introducción

El sistema virtual de archivos (VFS):

Linux soporta varios sistemas de ficheros
Nativos: Ext, Ext2 y Ext3
MS-DOS: FAT16, FAT32
Otros: Minix, etc.

Para permitir a los procesos un acceso uniforme a los archivos, el núcleo posee una capa lógica, el VFS, para ofrecer una interfaz de llamadas al sistema independiente del sistema de archivos montado y gestionar dichos sistemas.

El Buffer-Cache:

Para acelerar la E/S a disco se utilizan listas de memorias intermedias en uso.

La lectura de un bloque de disco se guarda en un buffer mientras se esté utilizando y no se necesite ese espacio de memoria. Los sucesivos accesos no se realizan sobre el disco, sino sobre el buffer.

Una modificación del buffer no se vuelca sobre la marcha a disco. Periódicamente, el proceso “update” llama a la primitiva “sync()” para forzar la reescritura de todos los buffers modificados → notable reducción de las E/S a disco.

16.2 El Buffer-Cache

El buffer-cache gestiona las memorias intermedias asociadas a los bloques de disco. La estructura buffer_head (<linux/fs.h>) define el descriptor de los buffers

Campo	Tipo	Descripcion
b_blocknr	unsigned long	Nº de bloque en el dispositivo
b_dev	kdev_t	Identificador del dispositivo lógico
b_rdev	kdev_t	Identificador del dispositivo físico
b_rsector	unsigned long	Nº del sector de inicio de bloque en el dispositivo físico
b_next	struct buffer_head *	Puntero al siguiente descriptor
b_this_page	struct buffer_head *	Puntero al siguiente descriptor cuyo contenido esta en la misma página de memoria
b_state	unsigned long	Estado del buffer
b_next_free	utruct buffer_head *	Puntero al siguiente descriptor libre
b_count	unsigned int	Nº de veces que ha sido referenciado ese bloque
b_size	unsigned long	Tamaño en bytes del bloque

b_data	char *	Puntero al contenido de la memoria intermedia
b_list	unsigned int	Lista en la que se encuentra el buffer
b_flushtime	unsigned long	Hora en la que debe escribirse a disco el contenido del buffer
b_lru_time	unsigned long	Hora en la que realizó el último acceso al buffer
b_wait	struct wait_queue *	Semáforo para controlar la concurrencia en los accesos al buffer
b_prev	struct buffer_head *	Puntero al descriptor anterior
b_prev_free	struct buffer_head *	Puntero al descriptor siguiente
b_reqnext	struct buffer_head *	Puntero al siguiente buffer que foma part de la misma petición de E/S

El estado (b_state) de un buffer puede ser cualquier combinación de las siguientes banderas:

Opción	Significado
BH_Uptodate	El buffer contiene datos válidos
BH_Dirty	El buffer ha sido modificado
BH_Lock	El buffer esta en uso y bloqueado
BH_Req	Si = 0, el buffer se ha invalidado
BH_Touched	El buffer ha sido utilizado recientemente
BH_Protected	El buffer no puede ser reutilizado
BH_FreeOnIO	El buffer esta asignado de manera temporal para efectuar una E/S y debe liberarse al terminar la operación

Los buffers se referencian mediante varias listas encadenadas según su estado y contenido.

Opcion	Significado
BUF_CLEAN	Lista de buffers no modificados
BUF_UNSHARED	Lista de buffers que ya no son compartidos
BUF_LOCKED	Lista de buffers a reescribir en disco
BUF_LOCKED1	Lista de buffer que contienen estructuras de control de los FS's (superbloques, i-nodes)
BUF_DIRTY	Lista de buffers modificados
BUF_SHARED	Lista de buffers compartidos

Funciones de gestión del Buffer-Cache

- Gestión del buffer caché → <linux/fs/buffer.c>.
- Interacción directa con la gestión de la memoria
- El contenido de los buffers deben colocarse en páginas de memoria principal.
- Los campos de la estructura mem_map que se corresponden de forma directa con el buffer-caché:
  - count: nº de referencias a dicha página
  - flags: estado de la página

- - buffers: puntero al primer buffer cuyo contenido está alojado en dicha página.

Categorías:

- Gestión propia de los buffers
- Realización de las E/S
- Modificación del tamaño de los buffers
- Gestión de los dispositivos
- Servicios para el acceso a los buffers
- Reescritura a disco
- Gestión de los clústers
- Inicialización de la estructura buffer-cache.

16.3 Gestión de los buffers

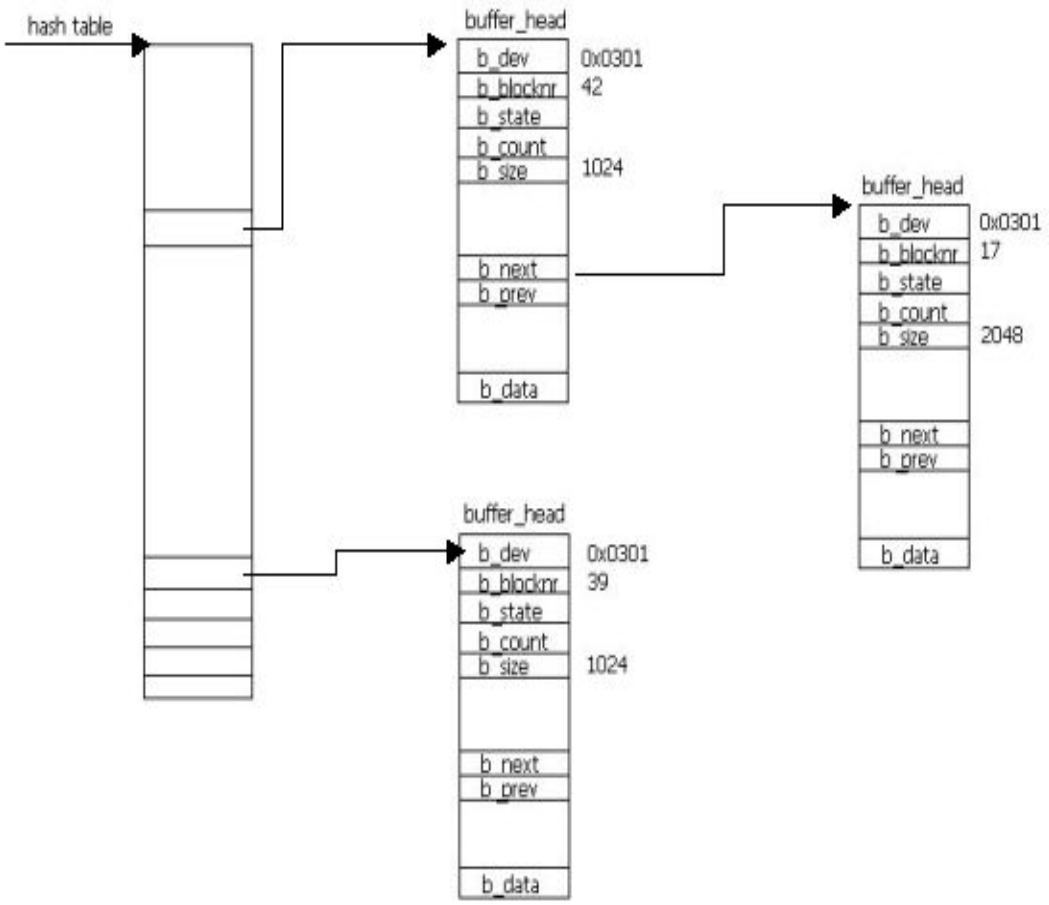
Cada sistema de ficheros tiene sus propias características (indexación, tamaño de bloque, etc).
Desde el inicio del sistema se tiene como mínimo una lista doblemente encadenada de buffers para cada sistema de fichero y tamaño de bloque montados.
Estas listas se actualizan constantemente cuando se asignan / liberan bloques, etc.
Para agilizar las búsquedas se tienen una o varias "listas de hash" por cada lista encadenada de buffers → similar al árbol AVL en las VMA's.

Se tienen otras listas encadenadas auxiliares ordenadas en base a diversos criterios:

- Lista de los más recientemente usados: LRU_list i
- Lista de los buffers no utilizados: unused_list i
- Lista de los buffers libres: free_list i
- Lista de los buffers reutilizables: reuse_list i

Las variables utilizadas para llevar a cabo esto son:

- hash_table: Tabla que contiene los punteros al primer buffer de cada una de las listas de hash en el sistema
- nr_hash: nº listas de hash para buffers en el sistema.



- **lru_list**: Tabla que contiene los punteros al primer buffer de cada lista LRU.
- **free_list**: Tabla que contiene los punteros al primer buffer disponible para cada tamaño de bloque.
- **unused_list**: Puntero al primer buffer no utilizado.
- **reuse_list**: Puntero al primer buffer no utilizado que debe reutilizarse.
- **nr_buffers**: nº de buffers asignados en el sistema.
- **nr_buffers_type**: Tabla que contiene el nº de buffers en cada una de las listas.
- **nr_buffers_size**: Tabla que contiene el nº de buffers para cada tamaño de bloque.

Funciones para la gestión:

- **__wait_on_buffer()** y **wait_on_buffer()**: Sincronizar los accesos concurrentes en modo núcleo a un mismo buffer.
- **_hashfn()** y **hash():** Implementan las listas de hash
- **remove_from_hash_queue(), remove_from_lru_list(), remove_from_free_list()**: eliminan un buffer de la lista.
- - **remove_from_queues()**: elimina todas las entradas para un buffer.
- **put_last_lru(), put_last_free()**: Insertan un buffer.
- - **insert_into_queues()**: Crea todas las entradas para un buffer.

- - **find_buffer()**: Búsqueda de un buffer en la lista de hash.
- - **refile_buffer()**: actualiza la lista LRU cuando se libera un buffer.

- - **put_unused_buffer_head():** libera un buffer no utilizado y lo añade a la lista. Llama a wake_up() sobre la cola "buffer_wait" para despertar a los procesos en espera de un buffer disponible.
- - **get_more_buffer_heads():** Se llama para asignar descriptores de buffers suplementarios. Es un proceso iterativo de tipo "garbage collector". En cada iteracion:

    - comprueba "unused_list" para ver si existen buffers asignados sin utilizar. Si es así devuelve el descriptor al llamador.
    - llama a **get_free_page()** para obtener una nueva región de memoria.
    - si falla, coloca al proceso llamador en la cola de espera buffer_wait, mediante la llamada a **sleep_on().**
    - si tiene éxito la página asignada se descompone en varios descriptores de buffer (dependiendo de su tamaño) y los añade a la correspondiente unused_list.

16.4 Funciones de Entradas / Salidas

Se encargan de crear los buffers, actualizar su contenido y hacerlos accesibles, así como tomar buffers temporales para las E/S y liberarlos automáticamente.

- create_buffers(): Crea nuevos buffers en una página de memoria. Asigna descriptores de memoria llamando a get_unused_buffer_head(), descompone la página en buffers de un tamaño especificado y coloca su dirección en el campo b_data de los descriptores de buffer.

- free_async_buffers(): Libera los buffers asociados con una página de memoria concreta. Recorre la lista encadenada y añade los buffers correspondientes a reuse_list para que pueden ser reutilizados (no se destruyen sobre la marcha).

- brw_page(): Encargada de realizar las lecturas/escrituras de disco hacia/desde una página de memoria.

- Toma nuevos descriptores temporales llamando a create_buffers() y se pone su estado a BH_FreeOnIO para indicar que deben ser liberadas despues de la operación de E/S.

- Busca en la tabla de hash si existe otro buffer asociado al mismo bloque en disco.

En caso de lectura de datos, el contenido se vuelca en la página de memoria.

En caso de escritura, la parte correspondiente de la página de memoria se vuelca al buffer temporal y ésta se marca como BH_Dirty.

- Al final del bucle se llama a ll_rw_block() para efectuar la operación de E/S.

__wait_on_buffer:

```
void __wait_on_buffer(struct buffer_head * bh) {
        /* obtenemos el descriptor de la tarea actual */
        struct task_struct *tsk = current;

        /* obtenemos el descriptor de la cola de espera pra esa tarea */
        /* DECLARE_WAITQUEUE definida en linux/wait.h */
        DECLARE_WAITQUEUE(wait, tsk);

        /* incrementamos el nº de accesos a este bloque de forma atomica */
        /* get_bh(), put_bh(bh) definidos en linux/fs.h */
        get_bh(bh);

        /* ponemos el proceso en la cola de espera y actualizamos su campo b_wait
*/
        add_wait_queue(&bh->b_wait, &wait);

        /* mientras no podamos acceder al buffer */

        do { /* ponemos la cola de procesos en espera por disco lista para ejecucion */
                run_task_queue(&tq_disk);

                /* actualizamos estado de la tarea */
                set_task_state(tsk, TASK_UNINTERRUPTIBLE);

                if (!buffer_locked(bh)) /* el buffer ya esta disponible */
                        break;
                        /* lanzar a ejecucion siguiente proceso */
                schedule();

        } while (buffer_locked(bh));

        /* actualizamos el proceso a su estado normal */
        tsk->state = TASK_RUNNING;

        /* lo eliminamos de la cola de espera */
        remove_wait_queue(&bh->b_wait, &wait);

        /* decrementamos nº de accedos al buffer */
        put_bh(bh); }
```

__refile_buffer:

```
static void __refile_buffer(struct buffer_head *bh) {
        int dispose = BUF_CLEAN;
```

```
        /* vamos comprobando el estado del buffer para insertarlo su lista */
        if (buffer_locked(bh)) dispose = BUF_LOCKED;
        if (buffer_dirty(bh)) dispose = BUF_DIRTY;
        if (buffer_protected(bh)) dispose = BUF_PROTECTED;

        /* cambio de lista */
        if (dispose != bh->b_list) {
                __remove_from_lru_list(bh, bh->b_list);
                bh->b_list = dispose;
                if (dispose == BUF_CLEAN)
                        remove_inode_queue(bh);
                        __insert_into_lru_list(bh, dispose); }
        }
```

create_buffers:

```
static struct buffer_head * create_buffers(
        struct page * page, unsigned long size, int async) {
                struct buffer_head *bh, *head;
                long offset;
        try_again:
        head = NULL;
        offset = PAGE_SIZE; /* maximo espacio de memoria a dividir */

        /* mientras haya hueco en la página ir tomando bloques */
        while ((offset -= size) >= 0) {
        /* primero miramos si existen descriptores sin usar en esa pagina */
        bh = get_unused_buffer_head(async);
        /* no tenemos memoria suficiente */
        if (!bh) goto no_grow;
        /* actualizamos la lista de descriptores*/
        bh->b_dev = B_FREE; bh->b_this_page = head; head = bh;

/* inicializamos los nuevos descriptores */
bh->b_state = 0; bh->b_next_free = NULL; bh->b_pprev = NULL;
atomic_set(&bh->b_count, 0); bh->b_size = size;

/* asocia una zona de la página con este descriptor de buffer */
set_bh_page(bh, page, offset);

/* los asociamos con sus respectivas listas */
bh->b_list = BUF_CLEAN; bh->b_end_io = NULL; }

/* devolvemos un puntero a la lista de nuevos descriptores */
return head;
no_grow: /* si se tomaron bloques */
if (head) { /* liberar todos los bloques tomados */
spin_lock(&unused_list_lock);
```

```
do { bh = head; head = head->b_this_page;
__put_unused_buffer_head(bh);
} while (head);

spin_unlock(&unused_list_lock);

/* despertar procesos en espera por bloques  para que puedan liberarlos */
wake_up(&buffer_wait);
}
/* fallo al complentar la operacion de IO */
if (!async) return NULL;

/* esperamos por que liberen bloques */
run_task_queue(&tq_disk);
current->policy |= SCHED_YIELD;
__set_current_state(TASK_RUNNING);
schedule();

/* volvemos a intentar tomar bloques */
goto try_again;
}
```

16.5 Modificación del tamaño del buffer cache

Grow_buffers

Se llama para ampliar el buffer cache.

1. Se asigna una página de memoria:
   `page = alloc_page(GFP_NOFS);`
2. Asigna las memorias intermedias
   `bh = create_buffers(page, size, 0);`
3. Habilita una exclusión mutua
   `spin_lock(&free_list[isize].lock);`
4. Inserta las memorias creadas en la lista de memorias no utilizadas.
   `free_list[isize].list = bh;`
5. Libera la exclusión mutua.
   `spin_unlock(&free_list[isize].lock);`
6. Se añade la página a la lista de LRU.
   `lru_cache_add(page);`
7. Se Incrementa el número de páginas.
   `atomic_inc(&buffermem_pages);`

Refill_freelist

Llena la lista de memorias intermedias disponibles para un tamaño dado.

1. Comprueba si el nº de memorias disponibles para ese tamaño es superior a 100. Si es así, considera que hay suficientes memorias disponibles y retorna.

```
if (free_shortage())
    page_launder(GFP_NOFS, 0);
```

2. Si no, llama a *grow_buffers* para asignar nuevas memorias intermedias. Si las asignaciones son suficientes, se termina.

```
if (!grow_buffers(size))
```

3. Si el buffer cache no puede ampliarse, se despierta al demonio bdflush.

```
wakeup_bdflush(1);
current -> policy |= SCHED_YIELD;
__set_current_state(TASK_RUNNING);
schedule();
```

16.6 Funciones de acceso a memorias intermedias

getblk

Se usa para obtener un descriptor de memoria intermedia correspondiente a un dispositivo y a un número de bloque.

1. Llama a *get_hash_table* para buscar la memoria intermedia y devuelve el resultado si la búsqueda es positiva.

2. En caso contrario, se llama a *refill_freelist*, y se lanza una nueva búsqueda llamando a *find_buffer*.

3. Si la búsqueda es positiva, el tratamiento se reinicia.

4. Si no, se asigna un descriptor de la lista de memorias no utilizadas.

bread

Obtiene la memoria intermedia correspondiente a un dispositivo y a un número de bloque.

1. Llama a *getblk* para obtener el descriptor de la memoria.

2. Llama a *ll_rw_block* para leer el contenido de la memoria desde disco.

3. Devuelve la dirección del descriptor del descriptor o NULL en caso de error.

sync_buffers

Reescribe las memorias intermedias modificadas en disco.

1. Efectúa hasta 3 pasadas de exploración de listas:
   1ª pasada: las memorias intermedias modificadas se reescriben de manera asíncrona en disco. Las memorias bloqueadas se obvian.

2ª y 3ª pasada: se pone en espera sobre las memorias intermedias bloqueadas con *wait_on_buffer.*

2. El nº de pasadas depende del parámetro de entrada *wait.*

sync_old_buffers

Reescribe el contenido de las memorias intermedias en disco.

1. Llama a *sync_supers* y a *sync_inodes* para reescribir los descriptores de sistemas de archivos e i-nodos en disco, todo ello en exclusión mutua mediante *lock_kernel()*.

2. Explora la lista de memorias intermedias modificadas y llama a *ll_rw_block()* para reescribirlas en caso de que no esté bloqueada y su fecha de escritura se haya alcanzado.

```
flush_dirty_buffers(1);
```

bdflush

Implementa el proceso *bdflush*. Este proceso se crea al inicializar el sistema**.**

1. Inicializa el descriptor del proceso.

```
spin_lock_irq(&tsk -> sigmask_lock);
flush_signals(tsk);
sigfillset(&tsk -> blocked);
recalc_sigpending(tsk);
spin_unlock_irq(&tsk -> sigmask_lock);
```

2. Entra en un bucle infinito. En cada iteración guarda en disco el contenido de las memorias intermedias modificadas cuya fecha de escritura se haya alcanzado.

```
flushed = flush_dirty_buffers(0);
```

3. Suspende el proceso *bdflush* si el número de memorias modificadas es inferior al permitido.

```
if (!flushed || balance_dirty_state(NODEV) < 0){
   run_task_queue(&tq_disk);
   interruptible_sleep_on(&bdflush_wait);
    }
```

16.7 Inicialización del buffer cache

buffer_init

1. Crea la tabla de Hash.

```
hash_table = (struct buffer_head **) __get_free_pages(...);
```

2. Inicializa sus enlaces

```
for (i = 0; i < nr_hash; i++) hash_table[i] = NULL;
```

3. Inicializa la lista de bloques libres

```
for (i = 0; i < NR_SIZES; i++){
     free_list[i].list = NULL;
          free_list[i].lock = SPIN_LOCK_UNLOCKED;
         }
```

4. Inicializa la lista LRU

```
for (i = 0; i < NR_LIST; i++) lru_list[i] = NULL;
```