[COL331] Assignment 3: Smart Block Driver

Hemanshu Garg — 2022CS11090

1 Introduction

This report provides a description and analysis of the smart block device driver implementation. The driver creates a virtual block device in Linux kernel with a write caching mechanism. This is a loopback block device (which means it we maintain it in memory) that provides read and write operations with asynchronous write operations/

2 Driver Overview and functionalities

The ”smartbdev” driver creates a RAM-based block device with a built-in write caching mechanism. The primary features include:

- Asynchronous writes.
- Configurable cache size (via module parameter and runtime procfs interface)
- Statistics tracking through procfs
- Cache flushing functionality.
- fifo writes to a thread’s own view.

3 Key Data Structures

3.1 Cache Entry Structure

The driver uses a cache entry structure to store write operations temporarily before committing them to the main device storage:

```c
struct cache_entry {
    struct list_head list;       /* List management */
    sector_t sector;             /* Starting sector of cached data */
    unsigned int size;           /* Size of cached data in bytes */
    void *buffer;                /* Pointer to cached data */
    atomic_t ref_count;          /* Reference counter */
    struct work_struct work;     /* Work structure for flush operations */
    struct smartbdev *dev;       /* Pointer to parent device */
};
```

Listing 1: Cache Entry Structure

3.2 Smart Block Device Structure

The main device structure contains all necessary components to manage the block device:

```c
struct smartbdev {
    spinlock_t lock;             /* Device data access lock */
    struct request_queue *queue;
    struct blk_mq_tag_set tag_set;
    struct gendisk *gd;
    u8 *data;                    /* Actual device data storage */

    /* Cache management */
    struct list_head cache_list;
    size_t used_cache;
    size_t cache_size;
    spinlock_t cache_lock;       /* Cache access lock */
    struct workqueue_struct *flush_wq; /* Workqueue for flush operations */

    /* Statistics */
    atomic_t write_count;
    atomic_t cache_hits;
    atomic_t flush_count;

    /* procfs entries */
    struct proc_dir_entry *proc_dir;
    struct proc_dir_entry *proc_stats;
    struct proc_dir_entry *proc_flush;
    struct proc_dir_entry *proc_cache_size;
};
```

Listing 2: smartbdev Device Structure

4 Driver Implementation

4.1 Module Initialization and Cleanup

The driver follows the standard Linux kernel module initialization pattern:

```c
static int __init smartbdev_init(void);
static void __exit smartbdev_exit(void);

module_init(smartbdev_init);
module_exit(smartbdev_exit);
```

Listing 3: Module Initialization and Cleanup Functions

During initialization, the driver:

1. Registers a block device major number
2. Initializes device data structures and locks
3. Allocates memory for the device storage
4. Sets up the work queue, tag, and request queue.
5. Creates procfs entries for statistics and cache size and flush.
6. Adds the disk to the system.

During cleanup, it reverses these operations, ensuring proper resource deallocation.

4.2 Request Processing

The driver implements a request processing mechanism to handle read and write operations:

```c
static blk_status_t smartbdev_queue_rq(struct blk_mq_hw_ctx *hctx, const struct
    blk_mq_queue_data *bd);
static int process_request(struct request *rq, unsigned long long *nr_bytes);
static int process_read_request(struct request *rq, unsigned long long *nr_bytes);
static int process_write_request(struct request *rq, unsigned long long *nr_bytes);
```

Listing 4: Request Processing Function Signatures

The driver handles I/O requests through the modern blk-mq interface. The request processing flow is:

1. Request arrives at `smartbdev_queue_rq`
2. Based on direction (read/write), the appropriate processing function is called
3. Reads are kept synchronous which flush the cache upto that point and read data off the disk.
4. For writes, data is added to the cache and scheduled for writing back to disc async.
5. The request is completed and the result is returned

4.3 Write Caching Mechanism

The write caching system is one of the key features of this driver:

```c
static int add_to_cache(struct smartbdev *dev, sector_t sector, void *buffer, unsigned int size
    );
static void flush_cache_entry(struct smartbdev *dev, struct cache_entry *entry);
static void flush_entry_work(struct work_struct *work);
static int flush_all_cache(struct smartbdev *dev);
```

Listing 5: Cache-Related Function Signatures

Write operations are cached before being committed to the main device storage:

1. Write request is received
2. Data is stored in a new cache entry
3. The cache entry is added to the cache list
4. A flush entry (writing only this new sector) is scheduled using a workqueue as a work item.
5. The flush operation will eventually write the data to the main storage
6. If the cache size limit is reached, older cache entries are flushed,

4.4 Procfs Interface

The driver provides a procfs interface for monitoring and configuration:

```c
static int smartbdev_stats_show(struct seq_file *m, void *v);
static int smartbdev_stats_open(struct inode *inode, struct file *file);
static ssize_t smartbdev_flush_write(struct file *file, const char __user *buffer,size_t count,
     loff_t *ppos);
static ssize_t smartbdev_cache_size_read(struct file *file, char __user *buffer,size_t count,
    loff_t *ppos);
static ssize_t smartbdev_cache_size_write(struct file *file, const char __user *buffer, size_t
    count, loff_t *ppos);
```

Listing 6: ProcFS Function Signatures

The procfs interface provides:

- `/proc/smartbdev/stats` - Shows device statistics (cache size, used cache, operations counts)
- `/proc/smartbdev/flush` - Triggers a manual cache flush when written to
- `/proc/smartbdev/cache_size` - Allows reading and setting the cache size

5 Key Implementation Aspects

5.1 Memory Management

The driver uses vmalloc for memory allocation, which is appropriate for large memory blocks:

- Device data storage: `vmalloc(NUM_SECTORS * SMART_SECTOR_SIZE)`
- Cache entries: `vmalloc(sizeof(struct cache_entry))`
- Cache buffers: `vmalloc(size)`

5.2 Concurrency Control

The driver uses several mechanisms for concurrency control:

- `spinlock_t lock` - Protects access to device data
- `spinlock_t cache_lock` - Protects access to cache list and metrics
- `atomic_t ref_count` - Tracks references to cache entries. This was important to use when stress testing was done.
- Workqueues - For asynchronous flush entry operations.

6 Testing/ Issues ecountered and resolved

Following tests were done and improvments realized.

- **Stress Testing the driver** 4000 threads were spawned to concurrently write and then immediately read of from different sectors of the disk. This revealed key race conditions, which were otherwise not visble on smaller tests. Basically when cache is fulled we flush the cache but if the work item for cache entries now try to cache entries it was leading to null pointer derefence.
  Solution: Using a reference counter. A standard pattern for mainting ownership of the pointer was employed. This solved key concurrency issues.
- **Cache size smaller than the amount of data edge case** This was resolved by making the writes synchronous and directly to the disk. The philosophy behind this is that as the cache size is configurable and say some process sets to it zero, that is indicative of disabling cache and thereby async writes.
- **FiFo writes** To a thread's own view its writes need to be FIFO as a standard. This was ensured by currently using singlethreaded work queue. (A better potential solution would be to manage writes to a sector through a pool and ensuring fifo among them)

References

[1] Linux Kernel Labs, *Block Device Drivers Lab — Create and Delete a Request Queue*, `https://linux-kernel-labs.github.io/refs/heads/master/labs/block_device_drivers.html#create-and-delete-a-request-queue`

[2] CodeImp, *sblkdev/device.c*, `https://github.com/CodeImp/sblkdev/blob/master/device.c`

[3] Ashay Shirwadkar, *ldd.c — LDD examples*, `https://github.com/ashayshirwadkar/ldd/blob/master/src/ldd.c`

[4] Bootlin Elixir, *proc_fs.h (v6.1.60)*, `https://elixir.bootlin.com/linux/v6.1.60/source/include/linux/proc_fs.h#L29`

[5] M. Kerrisk, *Linux Device Drivers, 3rd Edition — Chapter 16: Block Devices*, `https://static.lwn.net/images/pdf/LDD3/ch16.pdf`

[6] M. Kerrisk, *Linux Device Drivers, 3rd Edition — Chapter 2: Introduction to Kernel Modules*, `https://static.lwn.net/images/pdf/LDD3/ch02.pdf`