COL331 Assignment 3

Anubhav Pandey
2022CS51136

April 30, 2025

Table of Contents

1 Introduction

This report details the design, implementation, and evaluation of a smart block device driver developed as part of Assignment 3 (Hard) for the Operating Systems course. Traditional block device drivers manage hardware devices and provide an interface between the kernel and user applications. Block devices store data in fixed-size blocks and are accessed through a buffer cache. This assignment required the development of a more advanced block device driver with features such as priority-aware asynchronous writes, request caching, process-aware flushing, runtime statistics, and dynamic cache resizing to improve performance and flexibility. The report will cover the entire development process, from the initial design to the final testing and debugging stages.

2 Problem Statement

The primary objective of this assignment was to implement a block device driver with the following key features:

1. **Priority-Aware Asynchronous Write Operations:** Implement non-blocking write operations that prioritize requests from higher-priority processes. This requires a mechanism to identify the priority of the process initiating the write request and to manage the requests accordingly.
2. **Write Request Caching:** Implement an in-memory cache for write requests. The cache size should be configurable by the user at module load time. This feature aims to reduce the number of actual write operations to the disk, thereby improving performance.
3. **Process-Aware Force Flush:** Provide a procfs interface that allows users to flush cached write requests for a specific process ID (PID). This is useful for ensuring data consistency for specific applications.

4. **Driver Statistics:** Expose key driver statistics, such as the current cache usage and the total number of write-to-disk operations, through a procfs interface. These statistics are useful for monitoring the driver's performance and behavior.

5. **Dynamic Cache Size Adjustment:** Allow the cache size to be adjusted at runtime without requiring the module to be unloaded and reloaded. This provides flexibility in managing system resources.

3 Design Overview

The smart block device driver is implemented as a loadable kernel module (LKM). It registers itself as a block device with a dynamically allocated major number and a single minor number. The driver simulates a disk using a contiguous memory buffer allocated using `vzalloc`. All write requests are managed through a priority-aware in-memory cache.

Key Data Structures

- `struct write_req`: This structure represents a cached write request. It includes the following fields:
  - `pid`: The process ID of the process that initiated the write request.
  - `prio`: The priority of the process.
  - `sector`: The sector number to be written.
  - `len`: The length of the data to be written.
  - `data`: A pointer to the data to be written.
  - `list`: A `list_head` structure for managing the cache as a linked list.
- `struct smartblock_dev`: This structure maintains the overall state of the block device driver. It includes the following fields:
  - `data`: A pointer to the memory buffer that simulates the disk.
  - `cache`: A `list_head` structure representing the cache.
  - `cached_bytes`: The number of bytes currently stored in the cache.
  - `cache_size`: The maximum size of the cache, in bytes.
  - `write_to_disk_count`: The total number of write operations performed on the disk.
  - `lock`: A spinlock used to protect access to shared data structures.
  - `gd`: A `gendisk` structure representing the block device.
  - `queue`: A `request_queue` structure for managing I/O requests.

Synchronization

A spinlock (`device.lock`) is used to protect access to shared data structures, such as the cache and the driver statistics. This ensures thread safety and prevents race conditions when multiple processes access the driver concurrently.

4 Implementation Details

Device Registration

The driver registers a block device with a dynamically allocated major number. The registration process involves the following steps:

1. Allocating a major number using `alloc_chrdev_region`.

2. Creating a `gendisk` structure using `alloc_disk`. The `gendisk` structure represents the block device to the kernel.
3. Setting up the `block_device_operations` structure, which defines the operations that can be performed on the block device. This structure includes pointers to functions such as `open`, `release`, and `ioctl`.
4. Registering the block device with the kernel using `add_disk`.

Request Handling

The driver uses the `blk-mq` framework for handling I/O requests. The `blk-mq` framework provides a scalable and efficient way to manage I/O requests in the kernel.

- **Write Requests:**
  1. When a write request is received, the driver first checks if the requested write size exceeds the cache limit. If it does, the cache is flushed, and the request is written directly to the device.
  2. Otherwise, the driver allocates a `write_req` structure to store the write request details.
  3. The driver then copies the data from the user space to the kernel space using `copy_from_user`.
  4. The `write_req` structure is then inserted into the cache, maintaining the order based on the process priority. Higher priority requests are placed at the beginning of the list.
  5. If the cache is full, the driver flushes higher-priority requests to make space for the new request.
- **Read Requests:**
  1. Before serving a read request, the driver checks the cache for any overlapping write requests.
  2. If there are any overlapping requests, the driver flushes those requests to ensure data consistency.
  3. The driver then reads the data from the device's memory buffer and copies it to the user space using `copy_to_user`.

Caching and Flushing

The driver implements a write-back caching strategy to improve performance. The cache is implemented as a linked list of `write_req` structures.

- The cache size is configurable via a module parameter.
- When a write request is received, it is added to the cache.
- The cache maintains the write requests in the order of process priority.
- Flushing is triggered when the cache is full or when a process-specific flush is requested via the procfs interface.
- During a flush operation, the driver iterates through the cache, writing the data to the device's memory buffer and removing the corresponding `write_req` structures from the cache.

Procfs Interface

The driver creates a directory and several files under `/proc/smart_block`. The procfs interface provides a way for users to interact with the driver and monitor its status.

- `stats`: A read-only file that displays the current cache size, the number of cached bytes, and the number of write operations performed on the disk.
- `flush`: A write-only file that allows users to trigger a flush operation for a specific process. Writing a PID to this file will flush all cached write requests associated with that PID.

- `resize_cache`: A write-only file that allows users to dynamically resize the cache. Writing the new cache size (in bytes) to this file will resize the cache.
- `cache`: A read-only file that lists all the pending write requests in the cache, including the PID, priority, sector, and length of each request.

Module Parameters

The driver uses module parameters to allow users to configure the driver at load time. The following module parameters are supported:

- `cache_size`: This parameter specifies the initial size of the cache, in bytes. The default value is 64KB.

5 Procfs Interface and User Interaction

The procfs interface is a key component of the driver, providing a means for user-space applications to interact with the driver and monitor its behavior. The driver creates a directory `/proc/smart_block` and several files within it to expose various functionalities.

`stats` File

The `stats` file provides read-only access to the driver's internal statistics. When a user reads this file, the driver formats the statistics into a human-readable string and copies it to the user-space buffer. The statistics include:

- `Cached bytes`: The current number of bytes stored in the cache.
- `Write-to-disk count`: The total number of write operations performed on the disk.

`flush` File

The `flush` file allows users to trigger a flush operation for a specific process. To use this feature, the user writes the PID of the target process to the file. The driver then iterates through the cache and flushes all write requests associated with that PID.

`resize_cache` File

The `resize_cache` file allows users to dynamically resize the cache. To resize the cache, the user writes the new cache size (in bytes) to the file. The driver then reallocates the cache with the new size.

1. The driver first validates the new cache size to ensure it is within acceptable limits.
2. The driver then allocates a new cache with the specified size.
3. The contents of the old cache are copied to the new cache.
4. The old cache is freed.

`cache` File

The `cache` file provides a read-only view of the current contents of the cache. When a user reads this file, the driver iterates through the cache and formats each write request into a human-readable string. The information displayed for each write request includes:

- `PID`: The process ID of the process that initiated the request.
- `Priority`: The priority of the process.
- `Sector`: The sector number to be written.
- `Length`: The length of the data to be written.

6 Testing and Demonstration

A comprehensive test script was developed to demonstrate and validate all the features of the smart block device driver. The test script performs the following actions:

1. **Module Loading:** The test script loads the driver module with a custom cache size using the `insmod` command.
2. **Write Requests:** The test script creates multiple processes with different priorities and issues write requests from each process. The write requests target different sectors of the block device.
3. **Data Verification:** The test script reads data from the block device to verify that the write operations were successful and that the data is consistent.
4. **Flush Operation:** The test script uses the `flush` file in the procfs interface to trigger a flush operation for a specific process. The test script then verifies that the write requests from that process have been flushed to the disk.
5. **Cache Resizing:** The test script uses the `resize_cache` file in the procfs interface to dynamically resize the cache. The test script then verifies that the cache size has been updated.
6. **Statistics Monitoring:** The test script reads the `stats` file in the procfs interface to monitor the driver's statistics, such as the cache usage and the number of write operations performed on the disk.

7 Challenges and Issues Faced

During the development of the smart block device driver, several challenges and issues were encountered:

1. **Cache Management:** Efficiently managing the cache to minimize the number of disk write operations while ensuring data consistency was a significant challenge. The driver had to be carefully designed to handle various scenarios, such as cache overflow, concurrent access, and process-specific flushes.
2. **Priority Handling:** Implementing priority-aware write operations required a mechanism to determine the priority of each process and to manage the write requests accordingly. The driver had to ensure that higher-priority requests were always processed before lower-priority requests.
3. **Procfs Interface:** Creating a robust and user-friendly procfs interface required careful attention to detail. The driver had to handle various error conditions, such as invalid input and file not found, and provide informative error messages to the user.
4. **Synchronization:** Ensuring thread safety was a critical concern, as multiple processes could access the driver concurrently. The driver used spinlocks to protect shared data structures and prevent race conditions.
5. **Testing and Debugging:** Testing and debugging kernel modules can be challenging, as it often requires specialized tools and techniques. The driver was tested extensively to ensure that it was stable and reliable.

8 Conclusion

The smart block device driver successfully implements all the required features, providing enhanced performance and flexibility compared to a standard block device driver. The driver uses a priority-aware caching mechanism, a procfs interface for runtime monitoring and control, and dynamic cache resizing. The development of this driver provided valuable insights into Linux kernel programming, device driver development, and kernel-user communication.

Appendix: Key Code Snippets

Priority-Based Write Request Insertion

```c
static void insert_write_req(struct write_req *new_req) {
    struct list_head *pos;
    struct write_req *req;
    // Insert by descending priority (higher prio first)
    list_for_each(pos, &device.cache) {
        req = list_entry(pos, struct write_req, list);
        if (new_req->prio < req->prio)
            break;
    }
    list_add_tail(&new_req->list, pos);
    device.cached_bytes += new_req->len;
}
```

Procfs Statistics Read

```c
static ssize_t stats_read(struct file *file, char __user *buf, size_t count,
   loff_t *ppos) {
    char tmp[128];
    int len = snprintf(tmp, sizeof(tmp),
        "Cached bytes: %zu\nWrite-to-disk count: %lu\n",
        device.cached_bytes, device.write_to_disk_count);
    return simple_read_from_buffer(buf, count, ppos, tmp, len);
}
```

Smart Block init

```c
    device.data = vzalloc(DEV_BYTES);
    spin_lock_init(&device.lock);
    INIT_LIST_HEAD(&device.cache);
    device.cache_limit = cache_size;
    device.cached_bytes = 0;
    device.write_to_disk_count = 0;

    static struct blk_mq_ops mq_ops = {
        .queue_rq = smartblock_queue_rq,
    };
    device.tag_set.ops = &mq_ops;
    device.tag_set.nr_hw_queues = 1;
    device.tag_set.queue_depth = 128;
    device.tag_set.numa_node = NUMA_NO_NODE;
    device.tag_set.cmd_size = 0;
    device.tag_set.flags = BLK_MQ_F_SHOULD_MERGE;
    device.tag_set.driver_data = NULL;

    ret = blk_mq_alloc_tag_set(&device.tag_set);
    device.queue = blk_mq_alloc_queue(&device.tag_set, NULL, &device);
    device.gd = blk_mq_alloc_disk_for_queue(device.queue, NULL);
        ...
    device.gd->major = register_blkdev(0, DEVICE_NAME);
        ...
    device.gd->first_minor = 0;
    device.gd->minors = 1;
    device.gd->fops = &smartblock_ops;
    device.gd->queue = device.queue;
    device.gd->private_data = &device;
    snprintf(device.gd->disk_name, 32, DEVICE_NAME);
    set_capacity(device.gd, NSECTORS);
        ...
    ret = add_disk(device.gd);
        ...
    device.flush_wq = create_singlethread_workqueue("smartblock_flush_wq");
        ...
    INIT_WORK(&device.flush_work, smartblock_normal_flush_work);
        ...
    proc_dir = proc_mkdir(DEVICE_NAME, NULL);
    proc_stats = proc_create("stats", 0444, proc_dir, &stats_fops);
    proc_flush = proc_create("flush", 0222, proc_dir, &flush_fops);
    proc_resize = proc_create("resize_cache", 0222, proc_dir, &resize_fops);
    proc_cache = proc_create("cache", 0444, proc_dir, &cache_fops);
```