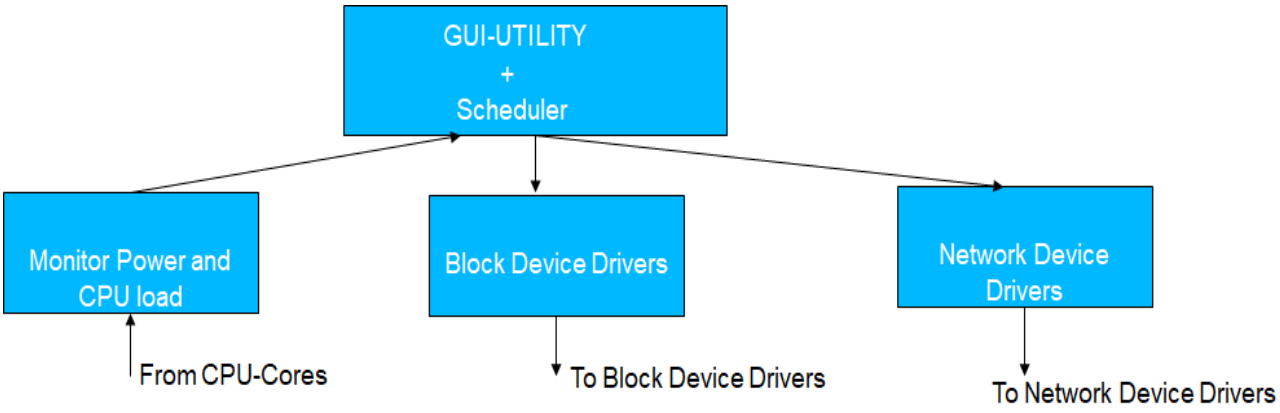
OS 533 Final Project Report
Block Device Drivers with Power and Performance Optimization

Suresh Srinivasan
Andre Mukhsia
Kamakshi Nagar
Pravallika Kavikondala
Neha Gadge

Architecture:

Block device drivers would implement the standard init, open, release, and request functions, which are invoked by the application when the drivers are loaded, opened and during read and write requests respectively. The standard scheduler for block I/O transfer would processes transfer data to block device by setting affinity to any CPU achieving maximum performance. To enhance the scheduler with power and performance tradeoff, we build our own scheduler which schedules appropriate processes based on a power and performance lookup table.



The existing load on the CPU-cores are monitored by the monitor process using PCM- Processor Counter Monitor- lib functions, and continuously update the scheduler with the current cpu-core load information. The scheduler will spawn threads based on the power performance lookup table and PCM metrics, and will bind the threads using sched_setaffinity API(s) to the appropriate CPU-cores. A good to have feature would be network device drivers, which would implement pre-hooks and post hooks to intercept ftp transfers based on the port numbers and transfer data to block I/O using scheduler and block device drivers developed.

Character Device Driver:

In exploration of block device drivers, we began exploring character device drivers. In a nutshell, char device drivers supports init, open, release, read, write and ioctl. Applications can load these device drivers using insmod command, and when insmod is executed the kernel

looks for the init function in the driver and is invoked appropriately. The char device drivers usually initializes the char device to the init state. Kernel invokes read or write functions in the char device drivers when an application requests for a read or write system calls. Char device drivers also support ioctl functions which are usually performs memory mapped registers or I/O configuration, which is invoked from the application space using ioctl system calls.

Block Device Driver:

In Linux, Block device drivers facilitates devices like disk drives to transfer random accessible data in fixed-size blocks. Linux kernel differentiates block device drivers from character devices drivers by providing different interfaces and their own particular challenges. Focus of the block layer design is centered on performance. Blocks are fixed size of chunk of data determined by kernel. Usually the block size are 4096 bytes but varies depending on the architecture and the exact file system used. A sector is a small block whose size is usually determined by the underlying HW. Usually linux kernel works with the world of 512-bytes sector, and expects to be dealing with 512-byte sector. If a HW uses a different sector size then kernel's sector size has to be scaled accordingly.

Block device drivers uses a set of registration interfaces to make their devices available to the kernel. Kernel supports function register_blkdev(unsigned int major, const char *name) for block device drivers to register their block device drivers. The parameter for the blk registration functions are major number that the device will use and the associated name, which the kernel displays in /proc/devices/. If a 0 is passed to the major number of the registration function call then the kernel assigns a new major number. Linux kernel also provides block device unregistration function unregister_blkdev(unsigned int major, const char *name) to unregister block device drivers

Block registration does not register devices by default, the kernel provides a separate registration interface that can be used to manage individual devices. Kernel provides block_device_operations structure, which is declared in <linux/fs.h>. The following are brief overview of block device drivers API(s):

1) int (*open) (struct block_device *, fmode_t): An Open system call to the applications to open the block device drivers in a read, write modes.
2) void (*release) (struct gendisk *, fmode_t) : releases the reference and frees up memory if any associated to the block device drivers.
3) int (*rw_page)(struct block_device *, sector_t, struct page *, bool);
4) int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long): Supports a direct block device control, which can be invoked from the user space.
5) int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
6) unsigned int (*check_events) (struct gendisk *disk,
   unsigned int clearing);

7) int (*media_changed) (struct gendisk *): calls supported for removable devices, the media method is called to see whether the media has been changed.
8) void (*unlock_native_capacity) (struct gendisk *);
9) int (*revalidate_disk) (struct gendisk *);
10) int (*getgeo)(struct block_device *, struct hd_geometry *);
11) /* this callback is with swap_lock and sometimes page table lock held */
12) void (*swap_slot_free_notify) (struct block_device *, unsigned long);
13) struct module *owner: Owner of the module usually the owner is the device drivers.

Unlike character device drivers, block device drivers do not support functions that actually perform read and write functions but in block I/O these operations are actually handled by request function.

Kernel supports users to perform disk registration to represent individual disk device. Kernel supports three functions and a data structure to perform disk registration and de-registration.

1) Struct gendisk
2) Struct gendisk *alloc_disk(int minors)
3) Void del_gendisk(struct gendisk *gd)
4) Void add_disk(struct gendisk *gd)

Gendisk structure, minimum fields used for our implementation:

1) Major: Device number used by the disk
2) First_minor: Used for how driver partition the device
3) Disk_name: 32 character name of the disk device, which will be used in proc/partitions and sysfs
4) Fops: The device operations supported by device
5) Queue: The request queue that will handle device operations for the disk
6) Private_data: Kernel provided storage for driver's use, used to store internal representation of the block device driver

The size of the device/ disk associated with the gendisk can be set using the function:

set_capacity(struct gendisk *disk, sector_t size)

Where disk is the address of our gendisk object and size is the size of the disk in Number of Sectors.

Firstly, a call to alloc_disk is required for the kernel to allocate and initialize a gendisk object to be used for disk registration, as drivers cannot allocate the structure on their own. The minor number refers to the number of partitions the disk use. The fields of the gendisk structure above should then be initialized. The device driver then should call the add_disk function to make the disk available to the system.

For de-registration, del_gendisk will remove references to the disk and cleanup the gendisk object. Another function called put_disk can be called to free/ deallocate the gendisk object but is optional.

Initialization:
For our block device driver, we first set memory for our internal structure representation of the device using memset, set the size field to the size of the device and allocate virtual memory for the device using vmalloc. We then call register_blkdev, passing our predefined major number '240' and name of device as parameters to register the device. Then, the device request queue and its spinlock is initialized and allocated by calling spin_lock_init, passing the address of the lock in our internal device representation, and blk_init_queue, passing our request processing function which fetch requests and filters read/ write requests and the initialized spinlock. We then set the logical block size for the queue the device can address by calling blk_queue_logical_block_size and save our internal representation of the device in the request queue's 'queue data' field which is a private storage reserved for device driver use. Done with the setting up the request queue, a call to alloc_disk allocates the gendisk structure object which fields we then initialize, and then calling add_disk adds the gendisk to the list of active disks.

Request function and Queues:
Request function of the block device driver are invoked whenever kernel believes it is time for read, write or other operations on the device. Every device has a request queue and a request function is associated with a request queue. This is because the actual transfers to and from a disk can take place far away from the time the kernel requests them, and because the kernel needs the flexibility to schedule each transfer at the most propitious moment. During the block device driver initialization a queue is created and the request function is associated with it. The queue is also guarded with a spin-lock. The invocation of the request function is entirely asynchronous with respect to the actions of any user-space process. Kernel provides NameOfDriver_request(struct request_queue *q) function for block device drivers to implement request functionality. Request queues keep track of outstanding block I/O requests but also play a crucial role in the creation of those requests. The request queue stores parameters that related to kinds of requests the device is able to service like maximum size, how many separate segments may go in the request, HW sector size, alignment requirements etc.

Scheduler:
Currently linux scheduler distribute the existing load on all the cores due to the fact that processes create to transfer data to block devices are set to any CPU affinity. In our case when the application wants to transfer data to/from a block disk, the existing scheduler would use all the cores to achieve maximum performance. We propose a new scheduler for block I/O data transfer wherein it can optimize based on power and performance. We propose to schedule number of threads proportional to the number of cores based on power and performance. We

use PCM (processor counter monitor) metric to monitor the load on the cpu. We use sched_setaffinity and sched_getaffinity to bind a process to a set of cpu(s), which is derived based on a power and performance Look up table. A typical lookup table would look like:

Power	Performance	No of Cpu to be used
0	FULL performance	All the cpu(s)
~25%	~75%	¾ * no of CPU(s)
~50%	~50%	½ * no of CPU(s)
~75%	~25%	¼ * no of CPU(s)
Full Power Saving mode	Least performance	One CPU or the lowest possible

In our cs533 scheduler, we provide a command line option to input the file to be transferred to block device. The scheduler schedules number of threads based on the aforementioned lookup table. The parent process will bind the threads to appropriate CPU(s) as per table using sched_setaffinity.

PCM (processor counter monitor) monitor process: PCM gives the number of instructions per cycle metric but unfortunately VM-wares do not let us set MSR register. Instead we tried on a desktop and was able to set MSR register and PCM gave the I(instructions)/C (cycles) ratio. The limitation was that the VM server on aws (amazon web services) did not let us set the MSR, and was one of the bottle necks in getting PCM working.

Issues Encountered and Efforts:

Deciding on the scope of capabilities and requirements of our Block Device Driver and simulated device. I.e.

- device type: whether the device is a RAM disk or relay disk.
- memory allocation function to use, which we went for the simplest vmalloc
    - kmalloc/ kzalloc for contiguous physical memory (more efficient, but prone to allocation failure),
    - page alloc for allocating pages,
    - vmalloc for allocating virtual memory (not contiguous in phys, higher probability of successful allocation).
- Decide whether needing to have multiple devices or a single device
- Decide the size of our disk
- Decide whether to have a request queue (primarily used for slower devices such as Hard disks) or no request queue (optimized use for faster devices i.e. with Random access capability such as RAM disks)
- Decide whether direct access of device memory through user program is required for the use case of the project.

Issues during implementation:

- Syslog flooded with messages, using up disk space resource.
- Issue with request not finishing and ended up blocking I/O of the device for all processes, causing processes to hang and bound CPU to always be at 100% load, requiring a soft reboot to fix.

Research needed on:

- Potential candidates for the device simulation and attempt to understand the standard structures required.
- How to implement Block Device Driver from reference textbook, Linux kernel documentation (which are partially obsolete as references used outdated kernel versions), and the definitions in the kernel header files the program depends on.
- Kernel changes to the required structure's fields and macro definitions, and implement a solution according to a specific recent Kernel version: 4.15.0

References:

Linux Device Driver 3rd Edition - Textbook - Retrieved from: https://lwn.net/Kernel/LDD3/
Device number reference - Retrieved from:
https://www.kernel.org/doc/Documentation/admin-guide/devices.txt
www.kernel.org
www.lwn.net