Linux Kernel Programming
The Block Layer

Pierre Olivier

Systems Software Research Group @ Virginia Tech

April 26, 2017

Virginia Tech

Outline

1. Block devices and the block layer
2. Buffers and buffer heads
3. The `bio` structure and request queues
4. IO schedulers

Virginia Tech

Outline

1. Block devices and the block layer
2. Buffers and buffer heads
3. The `bio` structure and request queues
4. IO schedulers

Virginia Tech

Block devices and the block layer

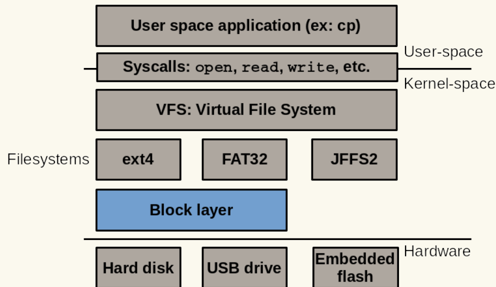
Blocks vs character devices

- There are 2 main types of devices in Linux:
  - **Character devices** are accessed sequentially as a stream of bytes, *byte by byte*
    - Examples: serial port, mouse, keyboard, etc.
    - Stream access: typing `test` on the keyboard result in the device sending `t`, `e`, `s`, then `t` to the driver
  - **Block devices** are accessed randomly, by **chunks**
    - Examples: HDD, SSD, CD/DVD, floppy disks, etc.
    - Random access: device can *seek* to a specific position, potentially non-sequential compared to the previous one

Block devices and the block layer

Blocks vs character devices (2)

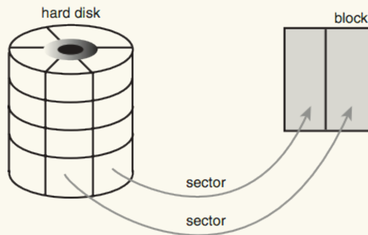
- Character device management is relatively simple and there is no subsystem entirely dedicated to them
- **Block devices are performance sensitive** (mostly used for storage)
  - There is a generic layer in the kernel dedicated to the management of block devices: **the block layer**



Block devices and the block layer

Block devices

- Minimum addressable unit in a block device: **sector**
  - Physical property of the device
  - Generally 512 bytes
  - Referred to as *sectors*, *hard sectors*, *device blocks*
- Software access the filesystem (partition) in **blocks**
  - Must a multiple of a sector (device limitation)
  - Must be a power of two and $<$ to a page size (kernel limitation)
  - Generally: 512 bytes, 1 kilobyte, 4 kilobytes
  - Referred to as *blocks*, *filesystem blocks*, *I/O blocks*



Outline

1 Block devices and the block layer

2 Buffers and buffer heads

3 The `bio` structure and request queues

4 IO schedulers

Virginia Tech

Buffers and buffer heads

- Read/written blocks are stored in memory in **buffers**
  - A buffer is an object representing one block in memory
  - A page can generally hold multiple buffers
  - A buffer has a descriptor, a **buffer head**
- `buffer_head` structure defined in `linux/buffer_head.h`:

```
1  struct buffer_head {
2    unsigned long         b_state;          /* buffer state flags */
3    struct buffer_head;   *b_this_page;     /* list of page's buffers */
4    struct page           *b_page;          /* associated page */
5    sector_t              b_blocknr;        /* starting block number */
6    size_t                b_size;           /* size of mapping */
7    char                  *b_data;          /* pointer to data within the page */
8    struct block_device   *b_bdev;          /* associated block device */
9    bh_end_io_t           *b_end_io;        /* I/O completion */
10   void                  *b_private;       /* reserved for b_end_io */
11   struct list_head      b_assoc_buffers;  /* associated mappings */
12   struct address_space  *b_assoc_map;     /* associated address space */
13   atomic_t              b_count;          /* use count */
14 }
```

Buffers and buffer heads

Buffer state

- State specified by the `b_state` field
  - Legal values stored in the `enum bh_state_bits` in `include/linux/buffer_head.h`:

- `BH_Uptodate`: contains valid data
- `BH_Dirty`: buffer is dirty
- `BH_Lock`: buffer is locked (disk I/O in progress)
- `BH_Req`: buffer is involved in an I/O request
- `BH_Mapped`: valid buffer mapped to an on-disk block
- `BH_New`: newly mapped buffer, not yet accessed
- `BH_Async_Read`: asynchronous read disk I/O in progress
- `BH_Async_Write`: asynchronous write I/O in progress

Virginia Tech

Buffers and buffer heads

Buffer state (2), usage count

- `BH_Delay`: delayed allocation, buffer is not associated to a block yet
- `BH_Boundary`: buffer forms the boundary of contiguous blocks, next block is discontinuous
- `BH_Write_EIO`: buffer incurred an I/O error on write
- `BH_Eopnotsupp`: buffer incurred a ”not supported” error
- `BH_Unwritten`: space for buffer has been allocated on disk but no data yet written
- `BH_Quiet`: suppress errors for this buffer

- Last item of the enum is `BH_Privatestart`:
  - Specifies the first bit usable by other code (drivers)
- Buffer usage count modified by `get_bh()` and `put_bh()`

Virginia Tech

Outline

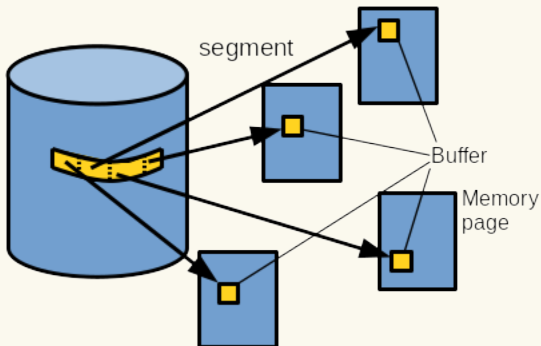
1. Block devices and the block layer
2. Buffers and buffer heads
3. The `bio` structure and request queues
4. IO schedulers

Virginia Tech

The bio structure and request queues

The bio structure

- Basic container for an active block I/O operation
- Uses *segments* to represent chunks of a buffer transferred to/from disk from/to memory
  - An individual buffer being divided into segments, it needs not to be contiguous in memory



The bio structure and request queues

The bio structure (2)

- struct bio defined in include/linux/blk_types.h

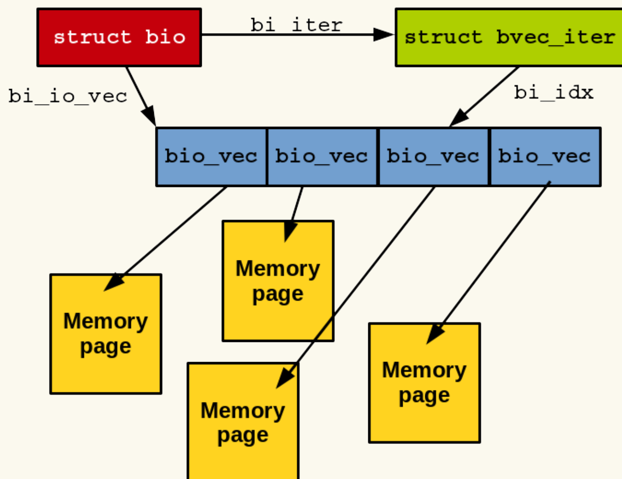
```
struct bio {
  struct bio          *bi_next;          /* list of requests */
  struct block_device *bi_bdev;          /* associated block device */
  unsigned short      bi_flags;          /* status and command flags */
  unsigned int        bi_phys_segments;  /* number of segments */
  struct bvec_iter    bi_iter;           /* vector iterator */
  unsigned int        bi_seg_front_size; /* size of front segment */
  unsigned int        bi_seg_back_size;  /* size of last segment */
  bio_end_io_t        *bi_end_io;        /* I/O completion method */
  void                *bi_private;       /* owner private data */
  unsigned short      bi_vcnt;           /* number of bio_vecs */
  unsigned short      bi_max_vecs;       /* maximum bio_vecs possible */
  atomic_t            __bi_cnt;          /* usage counter */
  struct bio_vec      *bi_io_vec;        /* bio_vec list */
  struct bio_vec      bi_inline_vecs[0]; /* inline bio vectors */
  /* ... */
};
```

- struct bvec_iter defined in include/linux/bvec.h:

```
struct bvec_iter {
  sector_t     bi_sector; /* target address on the device in sectors  */
  unsigned int bi_size;   /* I/O count */
  unsigned int bi_idx;    /* current index into bi_io_vec */
  /* ... */
};
```

The bio structure and request queues

The bio structure



The bio structure and request queues

I/O vectors

- I/O vectors represented by bio_vec structures, composing the bi_io_vec array (representing the full buffer)
- Defined in include/linux/bio.h:

```
struct bio_vec {
  /* pointer to the target physical page: */
  struct page *bv_page;
  /* length in bytes of the buffer: */
  unsigned int  bv_len;
  /* offset inside the page where the buffer resides: */
  unsigned int  bv_offset;
};
```

The bio structure and request queues

Request queues

- Block devices maintain **request queues** to store pending I/O requests
- Request queues are represented by the `request_queue` structure (`include/linux/blkdev.h`)
- Requests are added to the queue by high-level code (ex: filesystem),
  - Requests are pulled from the queue by the block device driver and submitted to the device
- A single request:
  - Represented by `struct request`
  - Can operate on multiple consecutive disk blocks, so it is composed of *one or more* `bio` objects

Virginia Tech

Outline

1 Block devices and the block layer

2 Buffers and buffer heads

3 The `bio` structure and request queues

4 IO schedulers

IO schedulers

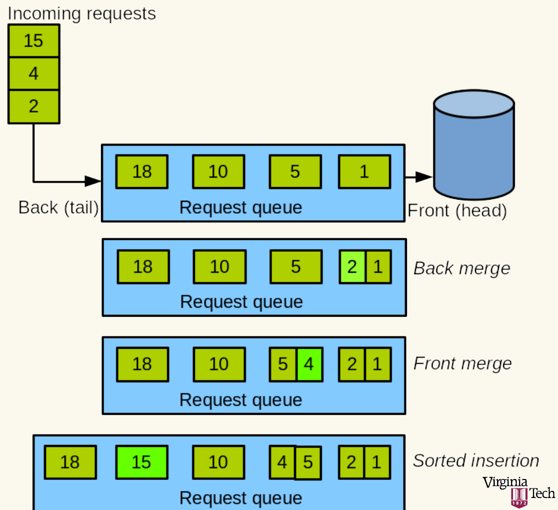
Presentation

- Directly sending requests to the disk as they arrive is sub-optimal:
  - Increase random accesses resulting in a lot of movement of the HDD head $\rightarrow$ *seeks*
  - The kernel tries to **reduce seeking** as much as possible
- The kernel combines and re-order I/O requests in the request queue:
  - **Merging**
  - **Sorting**
- Rules for merging and sorting are defined by the I/O scheduler
  - Multiple I/O scheduler models implemented in Linux
- The I/O scheduler *virtualizes* the disk as the process scheduler virtualizes the CPU

IO schedulers

The Linux elevator

- **Linus Elevator**: default in 2.4, replaced in 2.6
- **Define where an upcoming request should be added into the queue:**
  1. ***Back/front merge***
  2. ***Sorted insertion***, performed only if no request already in the queue is older than a give threshold $\rightarrow$ does not efficiently prevents starvation



IO schedulers

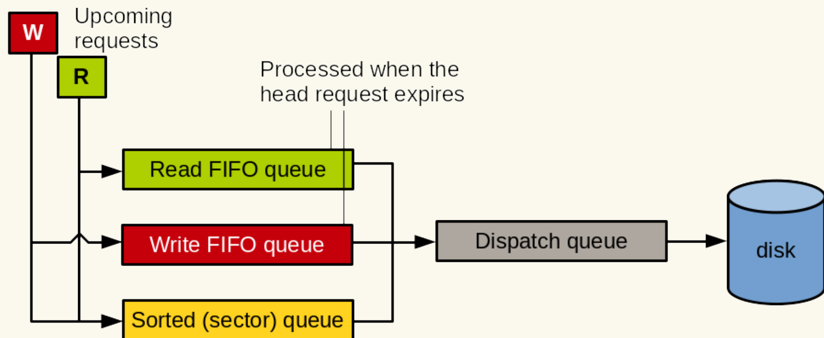
The deadline IO scheduler

- Problems with Linux Elevator:
  - A stream of requests to an on-disk specific location can starve other requests
  - **Write starving reads** issue
    - Contrary to reads, write are asynchronous from the application standpoint
    - Read latency is important for the system → **read starvation must be minimized**
- The **deadline scheduler** tries to provide fairness while maximizing the global throughput
- Implemented in `block/deadline-iosched.c`

IO schedulers

The deadline IO scheduler (2)

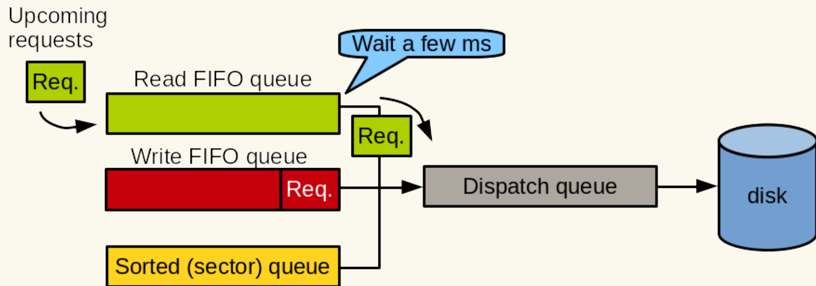
- Each request is given an expiration time, the **deadline**:
  - Reads: `now + .5s`
  - Writes: `now + 5s`



IO schedulers

The anticipatory IO scheduler

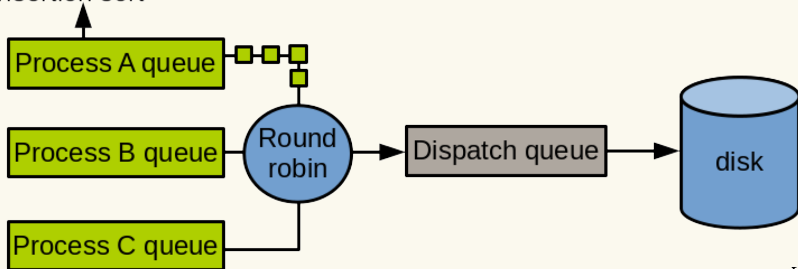
- **Anticipatory IO scheduler**
  - Dedicated to solve *deadline* throughput issues on certain scenarios
  - Removed in 2.6.18, replaced by CFQ



IO schedulers

The complete fair queuing IO scheduler

- **Completely Fair Queuing** I/O scheduler (CFQ)
  - Per-process request queues
  - `block/cfq-iosched.c`



IO schedulers

The noop IO scheduler

- **Noop** I/O scheduler
  - Does not perform anything in particular apart from merging sequential request
  - Used for truly random devices such as flash cards
  - `block/noop-iosched.c`

Virginia Tech

IO schedulers

IO scheduler selection

- I/O scheduler model can be selected at boot time as a kernel parameter: `elevator=<value>`
- `value` can be:
  - `cfq` for the completely fair queuing I/O scheduler
  - `deadline` for the deadline scheduler
  - `noop` for the noop scheduler