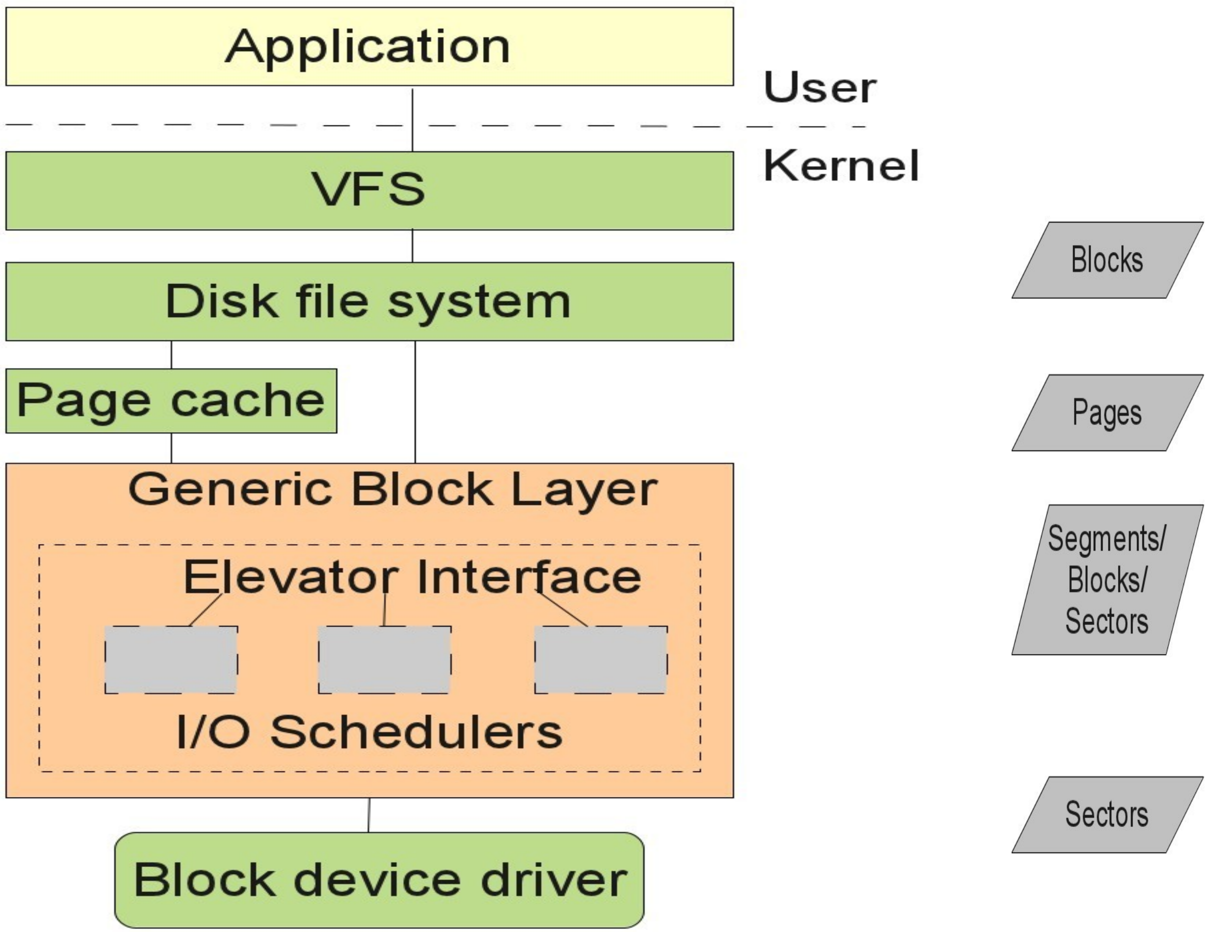
# The Journey of an I/O request through the Block Layer

**Suresh Jayaraman**
**Linux Kernel Engineer**
**SUSE Labs**
**sjayaraman@suse.com**

# Introduction

- Motivation
- Scope
  - Common cases
  - More emphasis on the Block layer
- Why should you listen to this talk?
  - Better understanding of I/O request handling
  - Help analyze I/O related problems
  - Just curious about finer details?

# Linux I/O Path



# Application I/O interface

- Read or write request
  - Stream I/O: fread()/fwrite()/fgets/fputs etc.
  - System calls: read() / write()
- Request to a file system or a device file
- For e.g.

***read(fd, buf, count);***

# System call handling (simplified)

- User mode process passes system call number
- CPU switches to kernel mode
- Finds the corresponding service routine via the system call dispatch table and invokes it.

# Relevant concepts

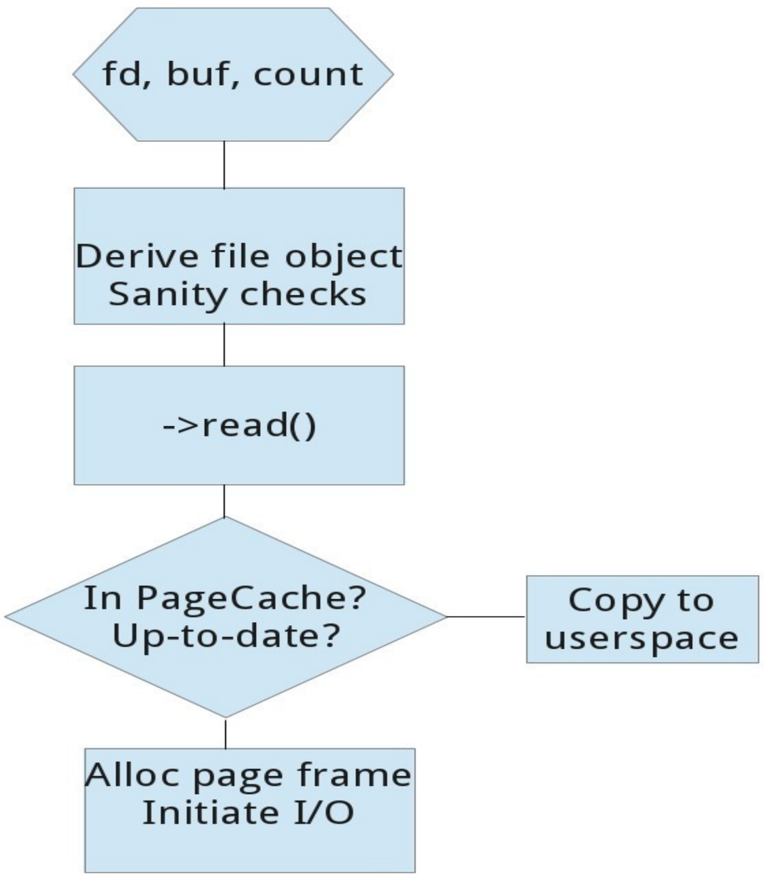
# Page cache

- In-memory store of recently accessed data
- Quicker subsequent access
- Page sized chunks (4k typically)
- Dynamic in size
- Auto-pruned (LRU) when memory is scarce

# Writeback and Readahead

- Writeback
  - Deferred writes
  - Data copied to buffer, marked as dirty and the write returns. The dirty buffer committed to disk later
  - Writeback triggers
    - When page cache gets too full
    - dirty buffer ages
  - Per backing device flusher threads
- Readahead
  - Adjacent pages read before they are actually requested
  - Enhances disk performance and system responsiveness (only in case of sequential access)

# I/O request handling at the File system Layer (1)



# I/O request handling at the File system Layer (2)

- Storage unit of data is blocks
- Determine the physical location of the data
  - File block number Vs Logical block number
  - Get file block number from file offset (offset/block_size)
  - File block number -> logical block number
- Allocate and prepare info needed by lower components (bio)
- Use generic block layer to submit requests

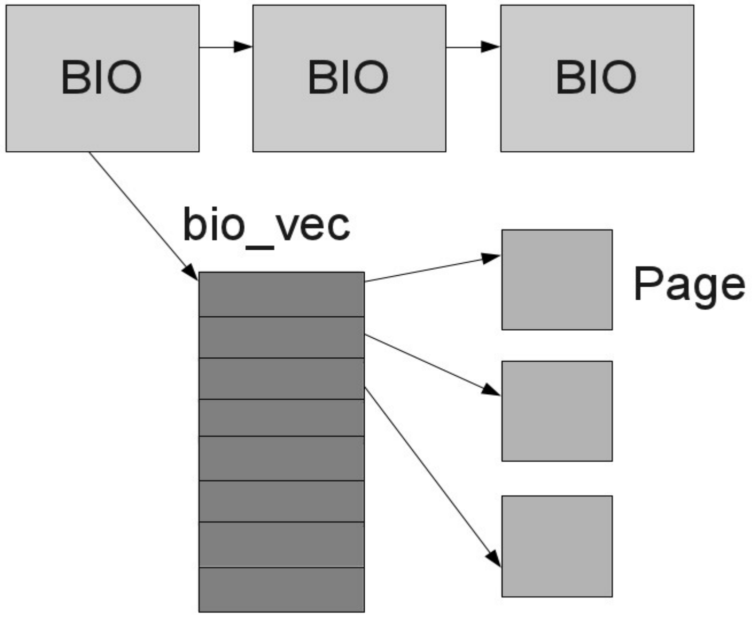
# Block Layer

# Block layer
## key concepts

- *bio* structure
- Request queues
- Partition remapping
- Device plugging and unplugging
- Merging and coalescing
- Elevator interface
- I/O schedulers

# *bio* structure

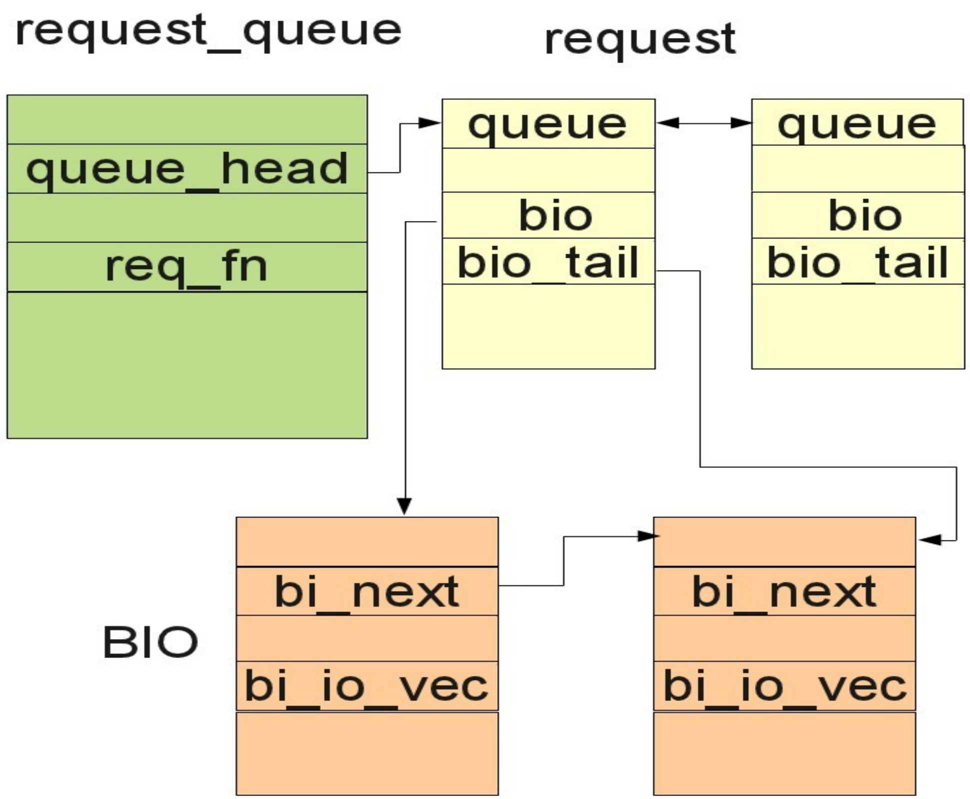
- Represents in-flight block I/O operations
- Scatter-gather I/O
  - I/O vectors (bio_vec)
- Has a pointer to block device
- ->end_io() callback to be used by device driver
- 1MB of data in a single bio (assuming 4k page size)



# Request queue

- Per device
- list of pending I/O requests for the device
- Each request has one or more bio's
- Knows which I/O scheduler handles its request
- Device driver specific *->request_fn*
- Methods for creating new requests and unplugging the device etc.

# Relationship between bio, request and request_queue



# Partition remapping

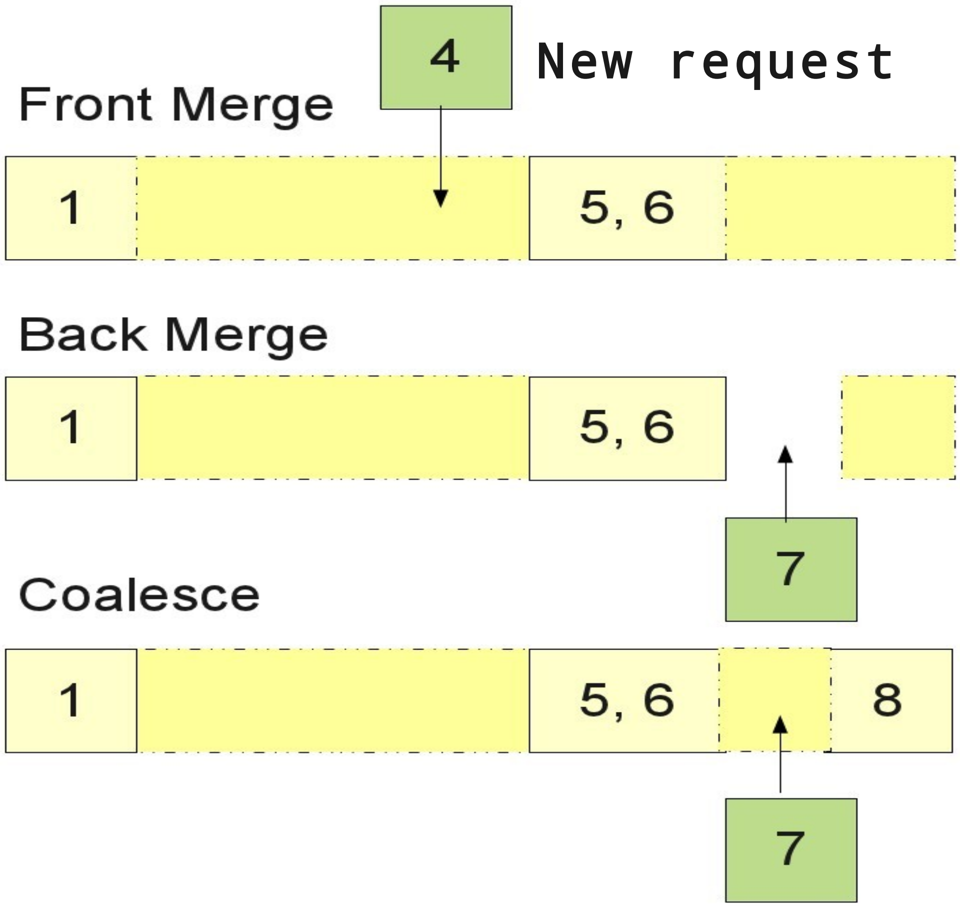
- Adjusts sector so that sector number is relative to the whole disk
- Sector 'n' of a partition starting at sector 'm' mapped to sector m + n of the block device.

  For e.g. read sector 256 of /dev/sda3
- Ensures correct area is read or written
- Sets block device to block device descriptor of whole disk
- Better scheduling decisions

# Plugging and unplugging

- Holds requests, allows build up of requests
- Why plugging?
- Implicit Vs Explicit block device plugging
  - How plugging happens?
  - Unplug triggers
    - Implicit: unplug_thresh, unplug timer elapses (usually 3msec)
    - Explicit: process finished submitting I/O

# Merging and coalescing



# Elevator interface

- Abstracts IO schedulers
- Provides merge searching and sorting functions
- Maintains a hash table of requests indexed by **end-sector** number
  - Back merge opportunities with constant time lookup
- No front merging
  - Up to the IO scheduler
- “One-hit” merge cache
  - stores the last request involved in a merge.
  - Checks for both front and back-merge possibilities

# I/O schedulers

- Algorithms for scheduling and re-ordering I/O operations
  - Overall throughput Vs starvation
- Registered with the elevator interface
- Implements scheduling policy via a set of methods (elevator_ops)
- Uses additional queues to classify and sort requests
- Different I/O schedulers: noop, cfq, deadline
- Scheduler can be switched during runtime

  /sys/block/<device>/queue/scheduler

# CFQ I/O Scheduler (default)

- The default scheduler
- Per-process sorted queues for sync requests
- Queues for async requests (1 per I/O priority)
- Round robin - picks a queue, lets it send requests for time-slice, picks another queue...
- Performs some anticipation

# I/O request handling at the generic block layer

- Gets request from the file system <bio, rw>
- Ensures bio doesn't extend beyond the device
- Gets the request queue of the block device
- Remaps the request if required
- If plugged, attempt merge to plug list
  - See if the request can be safely merged
    - Different data direction?
    - Same device?
  - Call elevator's allow_merge_fn

# I/O request handling at the Elevator layer

- Find merge opportunities in One-hit cache
- Find a potential merge and attempt merge
- If merge unsuccessful
  - Call I/O scheduler specific merge_fn

# I/O request handling at the I/O scheduler

- Elevator layer return merge possibilities or lack of
- Check if the request is merge-able
- I/O scheduler specific ->merged_fn method gets invoked
- If not merge-able, allocate a new request instance, fill it with data from bio
- Adds request to plug list if device is plugged
- If not, adds request to the request list
- Kicks the device queue by invoking the ->request_fn for that device queue

# Handling at the device driver (simplified)

- Device driver specific ->request_fn
- Hardware-specific task
- Typical sequence
  - Reads requests sequentially from the request queue
  - Starts data transfer
    - Sends READ/WRITE commands to disk controller
    - The disk controller raises an interrupt to notify the device driver
  - Waits for IO completion, invokes end_io callback
  - Block layer does the cleanup or wakes up waiting process

# Summary

- We explored different phases an I/O request goes though
- How the request gets handled in different layers
- Learned some concepts that are needed to understand block subsystem
- But... The Devil is actually in the “code” :)

# Q & A

Thank you.

# Different IO schedulers

- Noop
  - Performs merging, but no sorting
  - Truly random access, Intelligent devices
    - e.g. flash, ramdisk etc.
- Deadline
  - assigns to each request a deadline
    - 500 ms for reads, 5s for writes?
  - Three queues: Sorted queue, Read FIFO & Write FIFO
  - Good for server workloads

# How different layers manage disk data (Units)

- I/O scheduler and block device drivers => sectors
- VFS and mapping layer and filesystems => blocks
- Block device drivers => segments
- Page cache/Disk cache => page
  - As controllers of hardware block devices transfer data in chunks of sectors
  - Sector: Smallest addressable unit, defined by the device (power of 2, usually 512 bytes)
  - Page: Fixed-length block of main memory that is contiguous in both physical and virtual memory addressing. Smallest unit of data for memory allocation performed by the OS.
  - Block: Smallest addressable unit, defined by the OS (power of 2, at least sector size, at most page size)
  - Buffer: Represents a disk block in memory
  - Buffer head: struct that describes a buffer

# Why SUSE Linux Enterprise Server?

## Alliance Solutions: Server

### MISSION-CRITICAL INTEROPERABILITY

The most **interoperable** Linux for powering **physical**, **virtual, and cloud mission-critical** workloads

### PERFECT GUEST

**Optimized** to deliver **superior performance** as a guest OS **on leading** cloud-ready **hypervisors**

### INDUSTRY SUPPORT

Broad industry support, with the **most certified ISV applications** and the **most hardware certifications.**

# Why SUSE?

## Engineering Excellence

**20+ YEARS**

of **LINUX ENGINEERING EXCELLENCE**

SLES was the first enterprise Linux distribution

### SAP

**70%+**

Instances of SAP applications running on Linux run on SUSE Linux Enterprise Server

### KERNEL DEVELOPMENT

**TOP 3**

Where employees consistently rank among contributors to Linux kernel. SUSE employs 400 Linux developers.

### MAINFRAMES

**80%+**

Instances of Linux running on mainframes run on SUSE Linux Enterprise Server

### BROAD PLATFORM COVERAGE

**5 ARCHITECTURES**

SUSE Linux Enterprise Server runs on five architectures, from x86 to mainframe

### MICROSOFT RECOMMENDED

The **ONLY** Linux recommended by Microsoft is SUSE Linux Enterprise Server

### HIGH PERFORMANCE COMPUTING

**50%+**

HPC on Linux uses SUSE Linux Enterprise Server

### UNIFIED SUPPORT

Available for SUSE Linux Enterprise Server and Red Hat Linux

# Why SUSE?

## Broad Ecosystem

**PARTNER ECOSYSTEM**

**5000+**

**MEMBERS**

### CERTIFIED HARDWARE

**13500+**

Hardware systems certified and supported on SUSE Linux Enterprise—more than any other Linux distribution

### SOLUTION PROVIDERS & SYSTEM INTEGRATORS

**3200**

Value-added dealers and resellers

### CERTIFIED APPLICATIONS

**8500+**

ISV certifications—more than twice as many as Red Hat

### TECHNOLOGY PARTNERS

**1300**

Independent software and hardware vendors

### INTEROPERABILITY

Part of company-wide mission

### CERTIFIED PRODUCTS

**2000+**

SUSE Linux Enterprise products certified

### TRAINING PARTNERS

**600**

**Corporate Headquarters**
Maxfeldstrasse 5
90409 Nuremberg
Germany

+49 911 740 53 0 (Worldwide)
+www.suse.com

Join us on:
www.opensuse.org