**CS 326**: Operating Systems

# Disk Scheduling and SSDs

Lecture 15

# Today's Schedule

- Disk Scheduling
- Scheduling Algorithms
- SSDs

# Today's Schedule

- **Disk Scheduling**
- Scheduling Algorithms
- SSDs

# Disk Scheduling

- The usual question we ask in this class: how do we **virtualize** hard disk drives?
- One part is the file system, as we've studied
- The other ingredient is **disk scheduling**
- If we are careful about how we make requests from various processes, we can improve performance
  - If not? We'll spend most of our time seeking around the disk and not actually doing anything!

# Disk Scheduling: Core Principles

- Passing I/O requests directly to the underlying hardware is inefficient
  - Layering violation: applications are required to understand hardware details
  - Coordination: spatial locality of requests not considered
- Instead, I/O concerns are handled by a variety of OS subsystems, including the **I/O Scheduler**

# Linux File System Layer

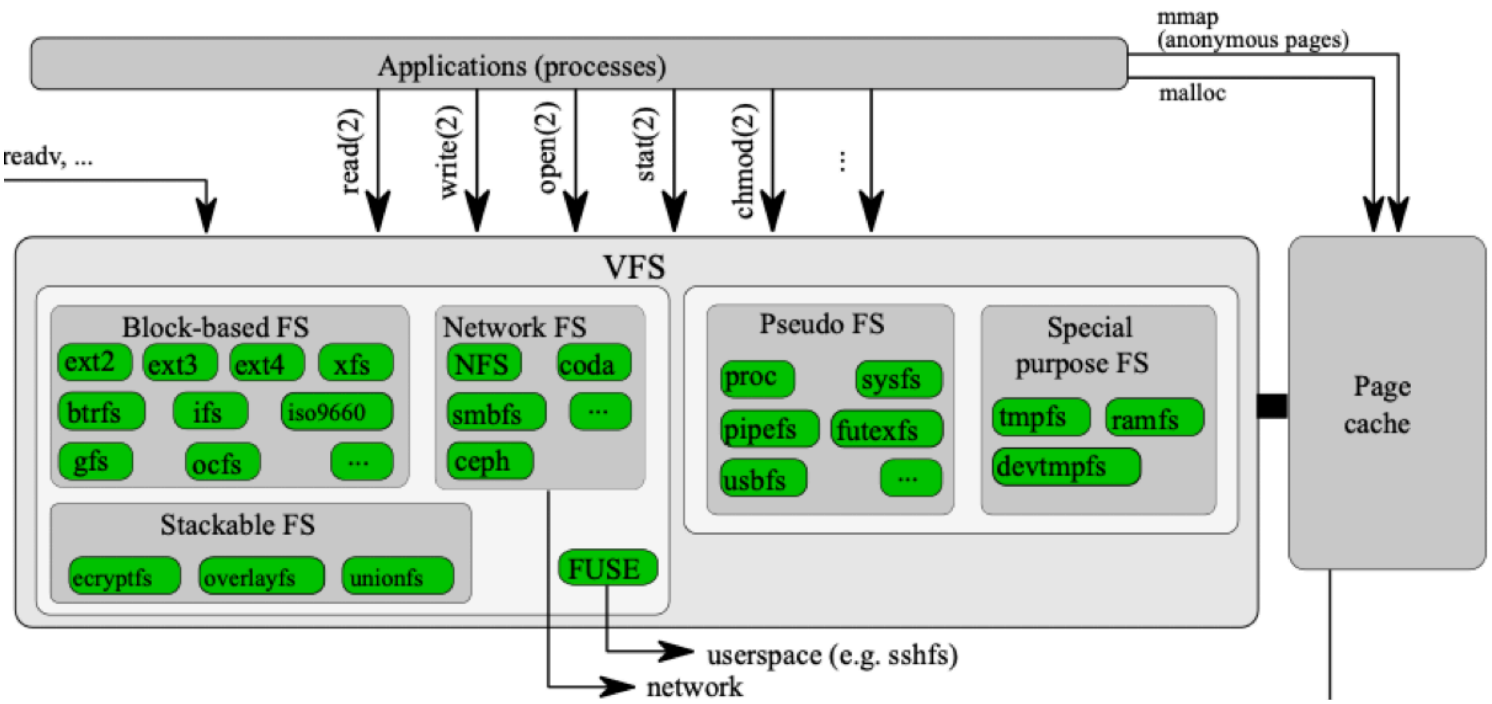



Image Credit: Thomas-Krenn AG, *Linux Storage Stack Diagram*

# Block I/O Layer

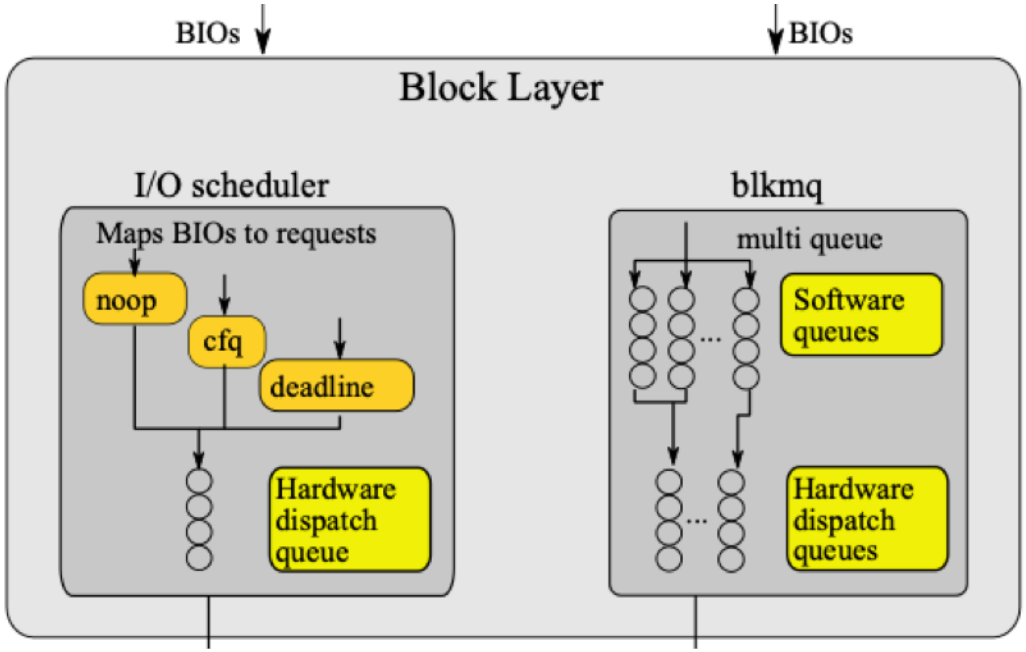



Image Credit: Thomas-Krenn AG, *Linux Storage Stack Diagram*

# Today's Schedule

- Disk Scheduling
- **Scheduling Algorithms**
- SSDs

# Algorithms

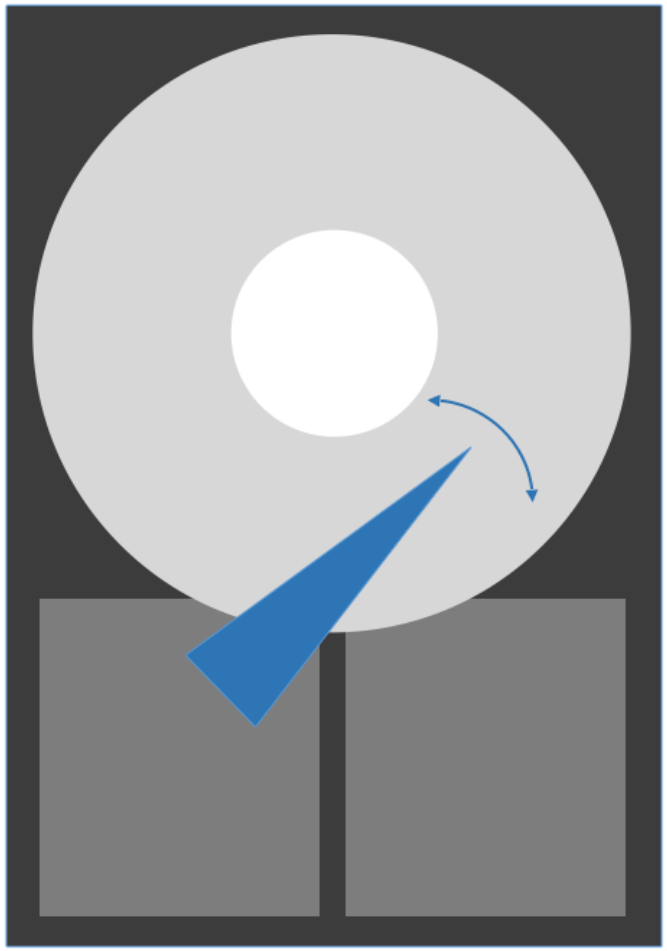
- FIFO Queue
- Elevator
- Deadline
- Anticipatory
- Completely Fair Queueing

# FIFO Queue

- The simplest way to handle incoming I/O requests is a first-in, first-out queue
  - Similar to submitting requests directly, but coordinated
- Enhancement: *request merging*
  - Rather than issuing 1000 requests to read a 4 MB file, merge logically contiguous requests together
- In Linux, this is implemented by the noop scheduler
  - Optimal for enterprise RAID systems, hardware schedulers/caches, VMs* and SSDs*

# The Elevator Algorithm

- Attempts to minimize changes in direction
  - Optimized for the physics of spinning disks
- Similar to riding an elevator!
  - Focus on overall throughput, not individual speed



# The Original Linux Scheduler

- One-way scan elevator: `elevator_linus`
  - Always scans in order of increasing LBAs (logical block addresses)
- Sequence numbers are assigned to requests to control latencies
  - Essentially forces requests to be flushed once enough accumulate (allows more disk head direction changes)
- Maintained as a doubly-linked list of I/O requests
  - Finding insertion/merge point: $O(n)$ – not scalable!

# The Deadline Scheduler [1/2]

- Both noop and the elevator algorithm emphasize throughput at the cost of latency
  - Can result in starvation
- Deadline scheduling: assign a deadline to each I/O request
- Requests are grouped by logical block address to optimize for disk head movements
- After servicing each group, the deadline queue is checked to see if any groups are being starved

# The Deadline Scheduler [2/2]

- Reads are prioritized over writes because processes often block while waiting for data from disk
- Best suited for multi-threaded workloads or small, random reads combined with sequential buffered writes (databases)
- Provides good performance for VMs and SSDs handling multiple workloads
  - Interestingly, most consumer SSDs benefit from request merging and LBA groupings

# Anticipatory Scheduling [1/2]

- A research scheduler, initially implemented in FreeBSD to deal with *deceptive idleness*
  - Processes often appear to have stopped doing I/O, but they're simply preparing for the next request
- Allows the disk to idle for a short period of time after servicing a request
  - Exploits spatial locality
  - 29-71% throughput improvement in disk-intensive Apache HTTP server workloads

# Anticipatory Scheduling [2/2]

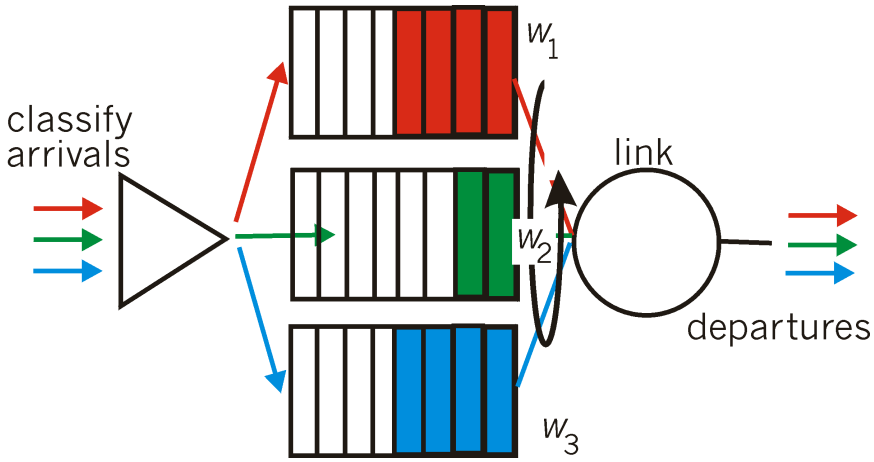
- Major issue: Requires application-specific tuning to get best performance
- Was the default Linux scheduler from kernel version 2.6.0 to 2.6.18
  - Replaced by CFQ (Completely Fair Queuing)

# Completely Fair Queuing [1/2]

- Based on ***stochastic fairness queuing***: each data flow is assigned to a fixed number of queues with a hash function
  - Frequently used to manage network traffic
- Queues are serviced on a round robin basis
- When the number of requests increases, collisions become much more likely
- In CFQ, each thread group in the system gets its own queue to avoid collisions

# Completely Fair Queuing [2/2]

- Supports a tunable idleness delay to handle deceptive idleness
- Low latency mode: also supports deadline queues
- Weighted fair queuing allows I/O prioritization



# MQ Schedulers

- All the previous schedulers we've discussed were designed for a single I/O queue
  - A design consideration from the spinning disk era
- To cope with modern storage (mostly SSDs) Linux switched to multi-queue (MQ) schedulers
  - CFQ was removed from the kernel in 2019, replaced with BFQ (budget fair queuing)
  - BFQ/CFQ can be configured to mimic anticipatory scheduling

# Choosing a Scheduler

- There are three schedulers available by default:
  - noop, mq-deadline, BFQ, and ***maybe*** kyber
  - **kyber**: noop + extra optimizations for fast SSDs (~1000 lines of code)
- To determine which scheduler you're using:
  - `cat /sys/block/${DEVICE}/queue/scheduler`
- Schedulers can be swapped at run time, on a per-device basis

# Today's Schedule

- Disk Scheduling
- Scheduling Algorithms
- **SSDs**

# Solid State Drives

- Unlike HDDs, SDDs have no moving parts
- More akin to the RAM in your PC rather than the spinning DVD/Blu-Ray drive (going out of style…)
- SSD **cells** store a charge
  - Multi-level cells store more than a single bit per cell
- **Banks** of these cells make up larger units of storage
- Most SSDs contain multiple banks
  - Combined via a microcontroller

# Blocks

- Each bank contains multiple **blocks**
  - ~128 – 256 KB
  - Each block contains multiple **pages** (sounds eerily familiar, huh?)
    - ~4 KB
- Most SSD operations are performed on the **block** level rather than individual cells
  - Most files take up more than 1 bit of space, after all

# Reading SSDs

- The great thing about SSDs is that we can read from anywhere, any time
- No worrying about the **location** of the data
- Higher performance SSDs support multiple reads at the same time

# Writing SSDs

- Writing gets a bit messier
- Before a write to a page can be performed, the *block* being written must be erased
- Since each block contains multiple pages, we have to store them somewhere first
  - We need to read the block into memory, update its state, and then write it
- **Write amplification**

# Boosting Write Performance

- As you can imagine, re-writing blocks is time consuming
- To improve the performance of write operations, the SSD controller will split the requests up
- Requests are handled concurrently by different cells

# F2FS: A Flash-Specific FS

- Given that many file systems and scheduling algorithms are designed for HDDs, SSD performance is not always optimal
  - There is no such thing as seek time, so do we need to combine requests that are close to one another?
  - What about multi-threaded access?
- Samsung created a new file system to help improve the performance of flash-based devices
  - Flash-Friendly File System (F2FS)

# Motivation

- For years, file systems have been designed with spinning disks in mind
  - These days the reality is much different
  - SSDs are commonly deployed in server environments
- Flash has its own set of considerations:
  - Erase-before-write
  - Write amplification
  - Limited cell life

# Other Efforts

- Apple recently migrated macOS, iOS, etc. devices to their new APFS
  - Has several optimizations/accommodations for flash-based storage
- As mentioned earlier, **kyber** is the answer on the Linux side of things
- Windows undoubtedly has optimized for SSDs, but of course development is proprietary
- We're still seeing huge progress being made with CPU **and** disk schedulers, even today!