Linux IO internals for PostgreSQL administrators in 2020

Ilya Kosmodemiansky (ik@dataegret.com)
Why this talk

- Linux is a most common OS for databases
- DBAs often run into IO problems
- Most of the information on topic is written by kernel developers (for kernel developers) or is checklist-style
- Checklists are useful, but up to certain workload
The main IO problem for databases for a long time was

- **How to maximize page throughput between memory and disks**
- Things involved:
  - Disks
  - Memory
  - CPU
  - IO Schedulers
  - Filesystems
  - Database itself

- IO problems for databases are not always only about disks
The main IO problem for databases for a long time was

• How to maximize page throughput between memory and disks

• Things involved:
  ▶ **Disks** - because latency of this part was very significant
  ▶ Memory
  ▶ CPU
  ▶ IO Schedulers
  ▶ Filesystems
  ▶ Database itself

• IO problems for databases are not always only about disks
Maximizing IO performance through maximizing throughput is easy up to certain moment.
Minimizing latency of IO usually is tricky.
With large adoption of proper SSDs, hardware latency dropped dramatically.
Because of high latency of rotating disks

- Database development was concentrated around maximization of throughput
- So did Linux kernel development
- Many rotating disks era IO optimization techniques are not that good for SSDs
PostgreSQL database

- DRAM
- Shared memory
- Page cache
- Disks
- WAL buffer
- WAL
- Datafile
- Database
- Linux
- User space
- Kernel space

<dataegret.com>
Key things about such workload

- Shared memory segment can be very large
- Keeping in-memory pages synchronized with disk generates huge IO
- WAL should be written fast and safe
- One and every layer of OS IO stack involved
Memory allocation and mapping

- CPU
- L1
- MMU
- TLB
- L2
- L3
- Memory
- Page table

Virtual addressing
Translation
Physical addressing
DBAs takeaways:

- Database with huge shared memory segment benefits from huge pages
- But not from transparent huge pages
  - Databases operate large continuous shared memory segments
  - THP defragmentation can lead to severe performance degradation in such cases
Freeing memory

Memory

- shared_buffers

Swap

- vm.swapiness
- vm.min_free_kbytes
- OOM-killer
- vm.panic_on_oom
- page-out
- reclaim
Page-out

- Page-out happens if:
  - someone calls fsync
  - 30 sec timeout exceeded (\textit{vm.dirty_expire_centisecs})
  - Too many dirty pages (\textit{vm.dirty_background_ratio} and \textit{vm.dirty_ratio})

- It is reasonable to tune \textit{vm.dirty_*} on rotating disks with RAID controller, but helps a little on server class SSDs
DBAs takeaways:

- `vm.overcommit_memory`
  - 0 - heuristic overcommit, reduces swap usage
  - 1 - always overcommit
  - **2 - do not overcommit** *(vm.overcommit_ratio = 50 by default)*
- `vm.min_free_kbytes` - reasonably high (can be easy 1000000 on a server with enough memory)
- **`vm.swappiness = 1`**
  - 0 - swap disabled
  - 60 - default
  - 100 - swap preferred instead of other reaping mechanisms
- Your database would not like OOM-killer
• *vm.panic_on_oom* effectively disables OOM-killer, but that is probably not the result you desire
• Or for a certain process: echo -17 > /proc/12465/oom_adj but again
Filesystems: write barriers

Journalated filesystem

Kernel buffer

Journal

Data

Journal entry

Data
DBAs takeaways:

- ext4 or xfs
- Disable write barrier (only if SSD/controller cache is protected by capacitor/battery)
IO stack (as it used to look like)
IO stack (as it used to look like)

- Database memory
- Page cache
- VFS
- EXT4
- Block device interface
- Disks
- Direct IO
- BIO Layer
- Request Layer
- Elevator/IO Scheduler
- Block device interface
- Disks

Legend:
- Operates pages
- Makes transition more efficient
- Operates cylinders/sectors
Elevators: before 2.6 kernel

- **Linus Elevator** - the only one in times of 2.4
- *merging* and *sorting* request queues
- Had **lots** of problems
Elevators: between 2.6 and early 3.*

- **CFQ** - universal, default one
- **deadline** - rotating disks
- **noop** or **none** - then disks throughput is so high, that it can not benefit from keen scheduling
  - PCIe SSDs
  - SAN disk arrays
Elevators: 3.13 and newer

- Effectiveness of **noop** clearly shows ineffectiveness of others, or ineffectiveness of smart sorting as an approach
- **blk-mq** scheduler was merged into 3.13 kernel
- Much better deals with parallelism of modern SSD - basically separate IO queue for each CPU
- The best option for good SSDs right now
- **blk-mq and NVMe driver is actually more than scheduler, but a system aimed to substitute whole request layer**
IO stack (with blk-mq)

- Database memory
- Direct IO
- VFS
- EXT4
- Page cache
- Block IO
- BIO Layer
- Kyber/BFQ IO schedulers
- blk-mq
- NVMe driver
- Disks
Good diagram on Linux IO stack

- https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram
- Regular updates
- Some things are difficult to draw, but it is a complex topic
Non Volatile Memory Express or NVMe

- Sets of standards, which helps to use modern SSDs more effectively
- For Linux it is first of all NVMe driver (or subsystem)
- Most common example of NVMe SSDs are PCIe NAND drives
- With NVMe v.5 (currently 3 is ready for production) can work up to 32GB/sec
- Are databases NVMe ready?
Latest development on new block layer

- IO polling
- New IO schedulers Kyber and BFQ (Kernel 4.12)
- IO tagging
- Direct IO improvements
- io_uring (Kernel 5.1)
Notes on Direct IO

- Currently PostgreSQL supports DirectIO only for WAL, but it is unusable on practice
- Requires a lot of development
- Very OS specific
- Allows to use specific things, like O_ATOMIC
- PostgreSQL is the only database, which is not using Direct IO
- Many people consider Direct IO as a dirty hack
- Looks like io_uring has better perspectives
Questions?

ik@dataegret.com