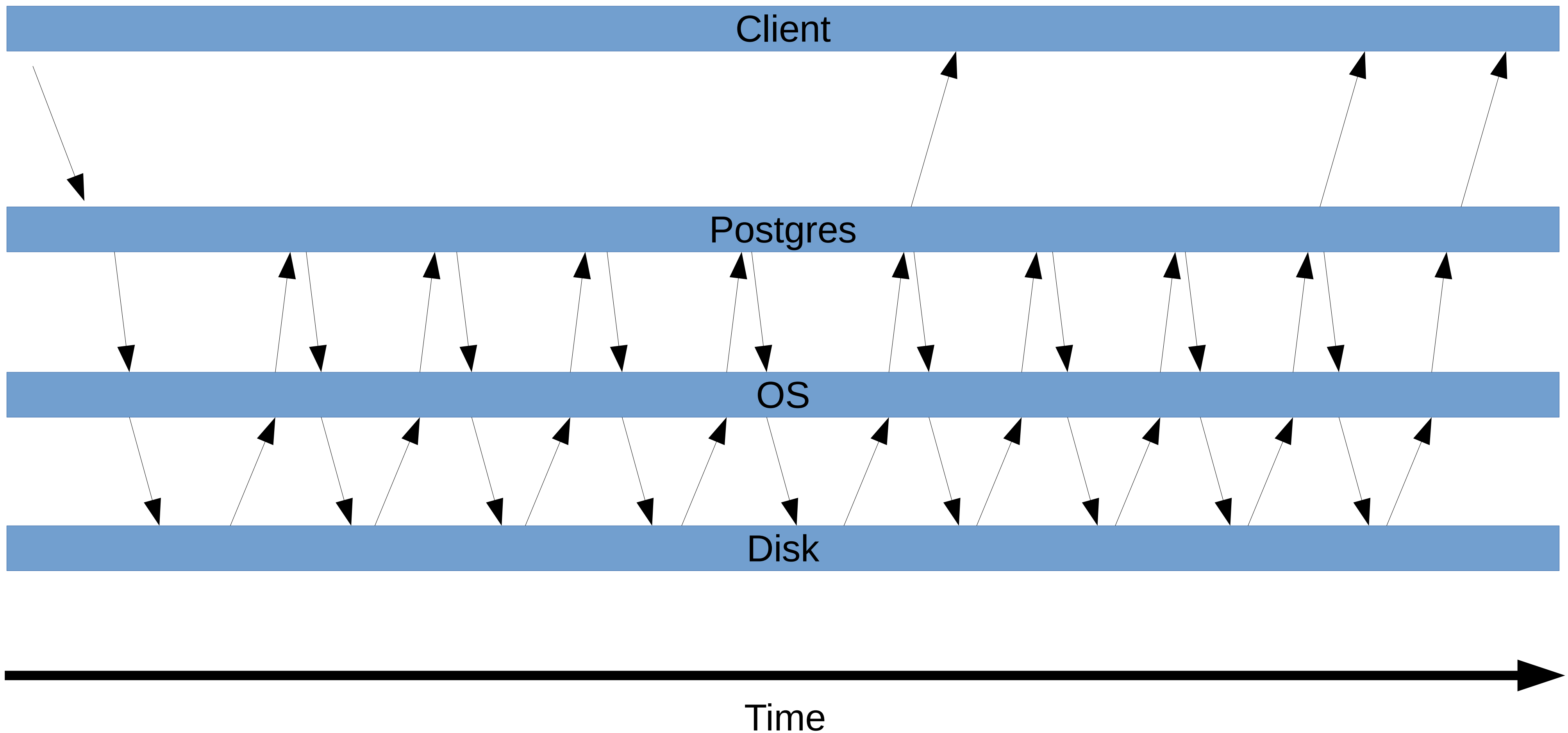
Asynchronous IO for PostgreSQL

(and probably also Direct IO)

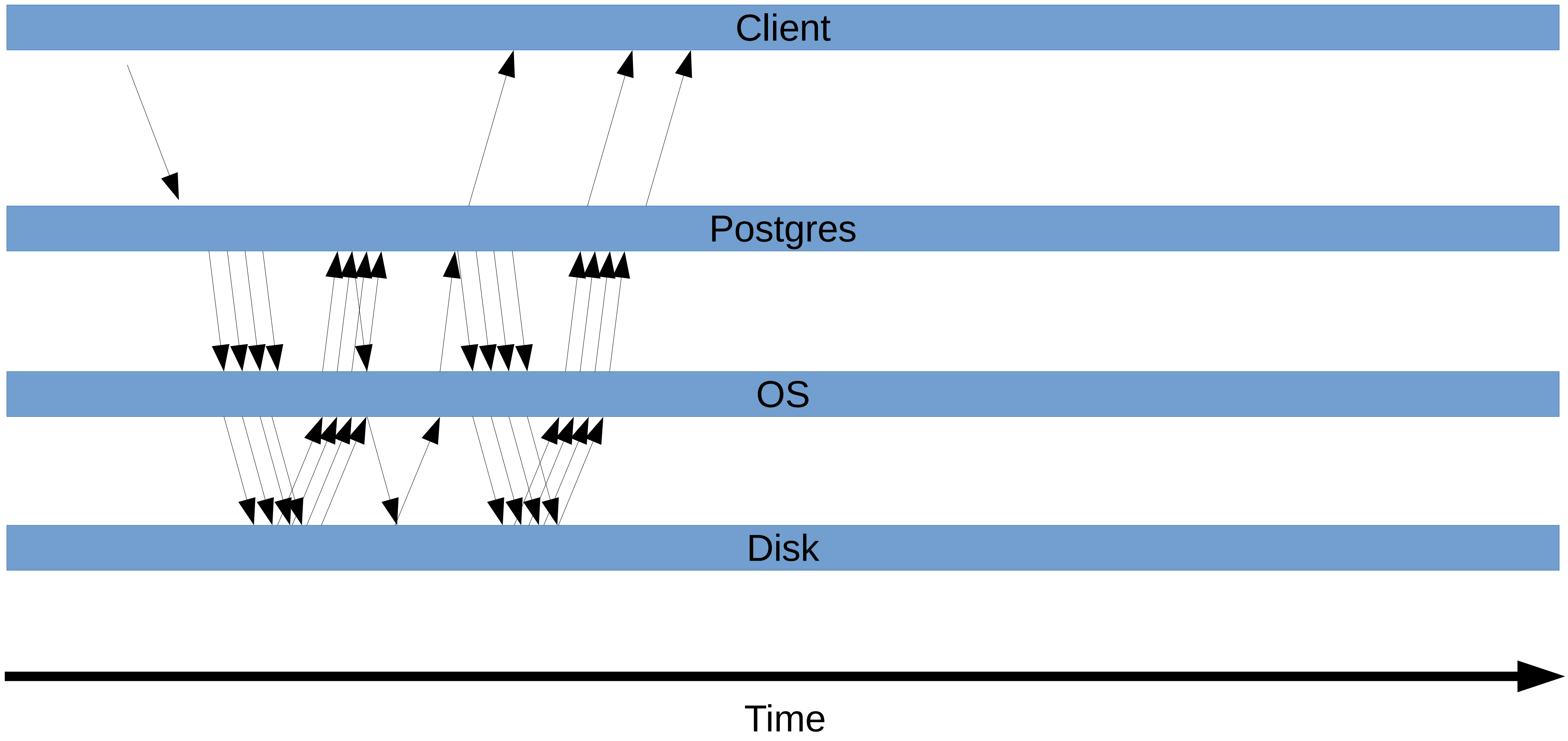
Andres Freund
PostgreSQL Developer & Committer

Email: andres@anarazel.de
Email: andres.freund@microsoft.com
Twitter: @AndresFreundTec
anarazel.de/talks/2020-01-31-fosdem-aio/aio.pdf

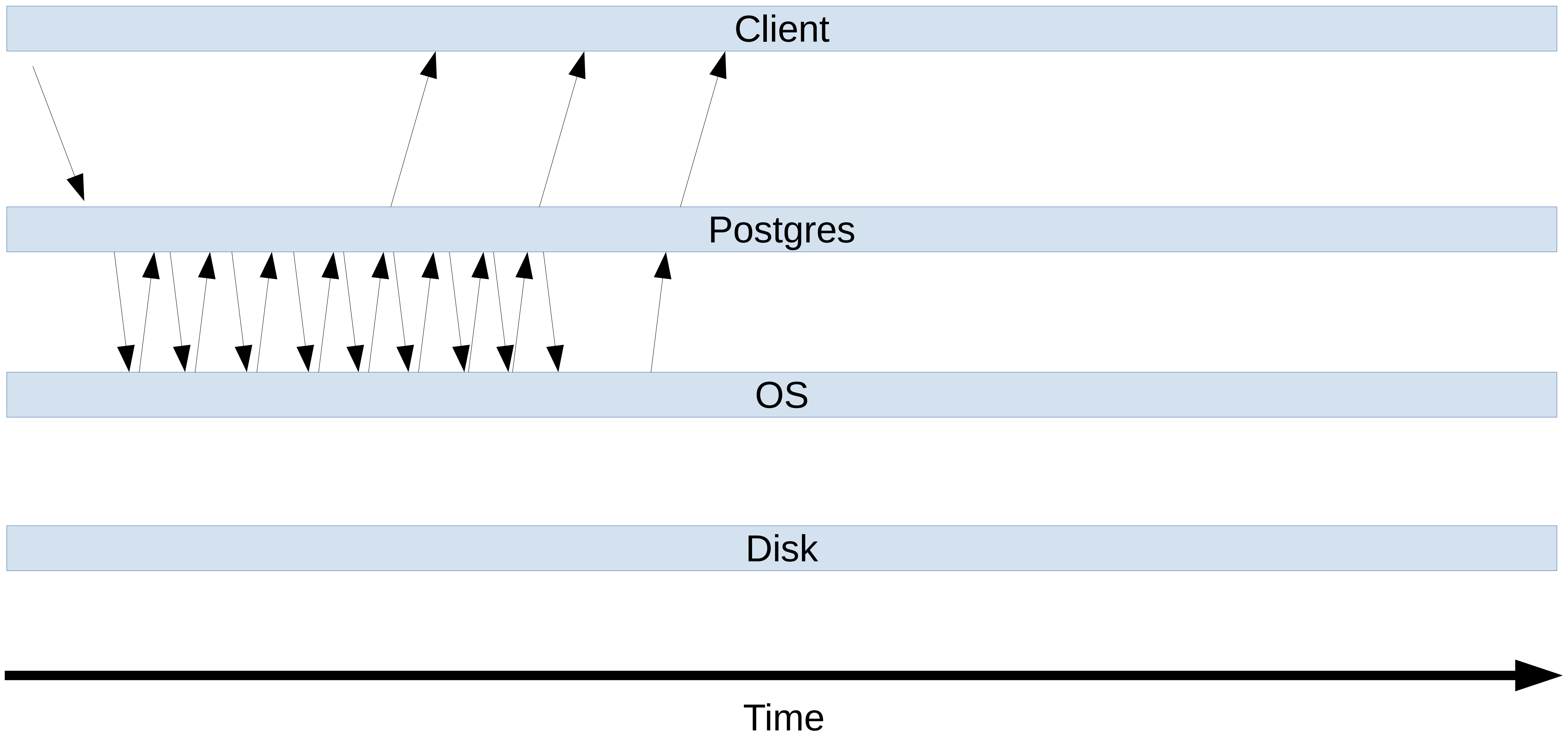
Reads: synchronous, not cached



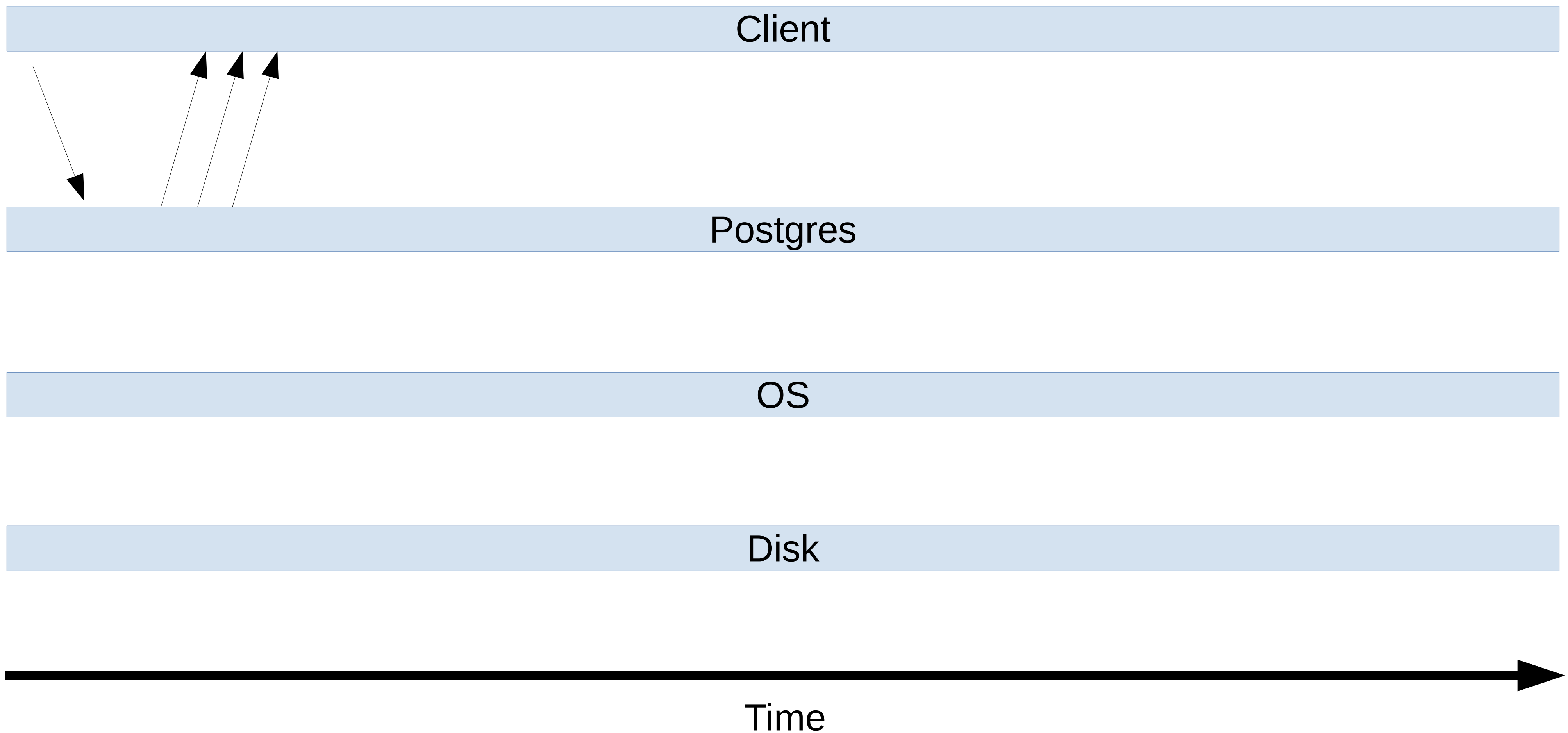
Reads: asynchronous, not cached



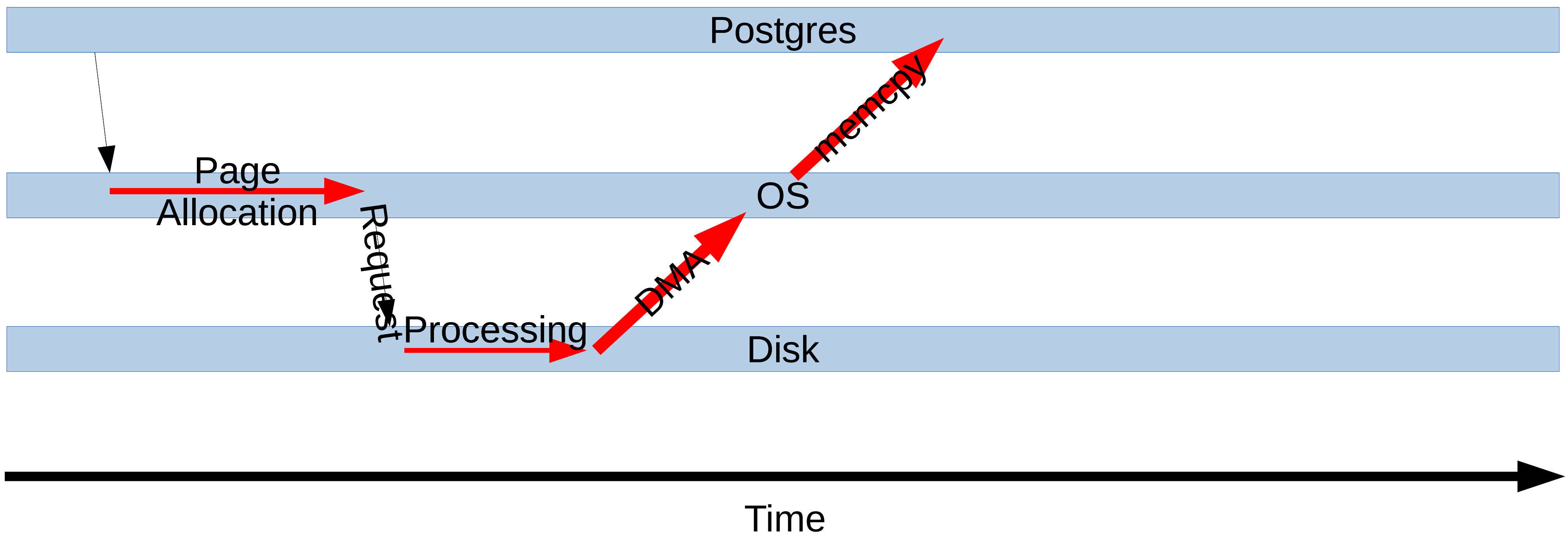
Reads: synchronous, OS cached



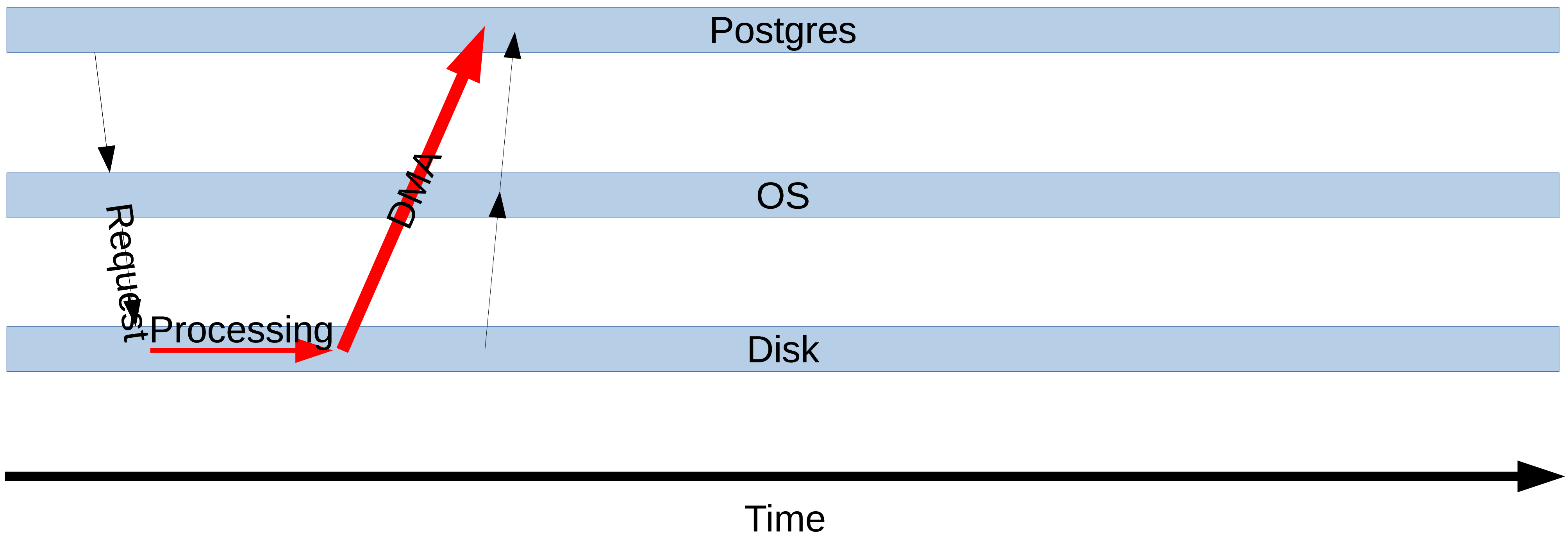
Reads: synchronous, postgres cached



Buffered Read



Non-Buffered Read



Hardware Trends:

- Massive throughput increases in commonly used storage
  - PCIe attached storage (NVMe SSDs)
  - massive arrays of disks (cloud block devices)
  - >3GB/s R/W for commodity prosumer hardware
- Massive parallelism increase
  - SSD: cannot be exploited through e.g. AHCI / SATA
  - cloud: actually talking to complicated storage array using many disks internally

Hardware Trends

- Latency:
  - PCIe SSDs: low microseconds (< 1000ns for some)
  - cloud: ~1-5 milliseconds
- Random writes:
  - SSDs: Noticable, but not hugely. May impacts lifetime
  - cloud: often basically not noticable, can be higher throughput for fast / large devices
- CPU & Memory:
  - Many more cores
  - Bandwidth per core **not** increasing

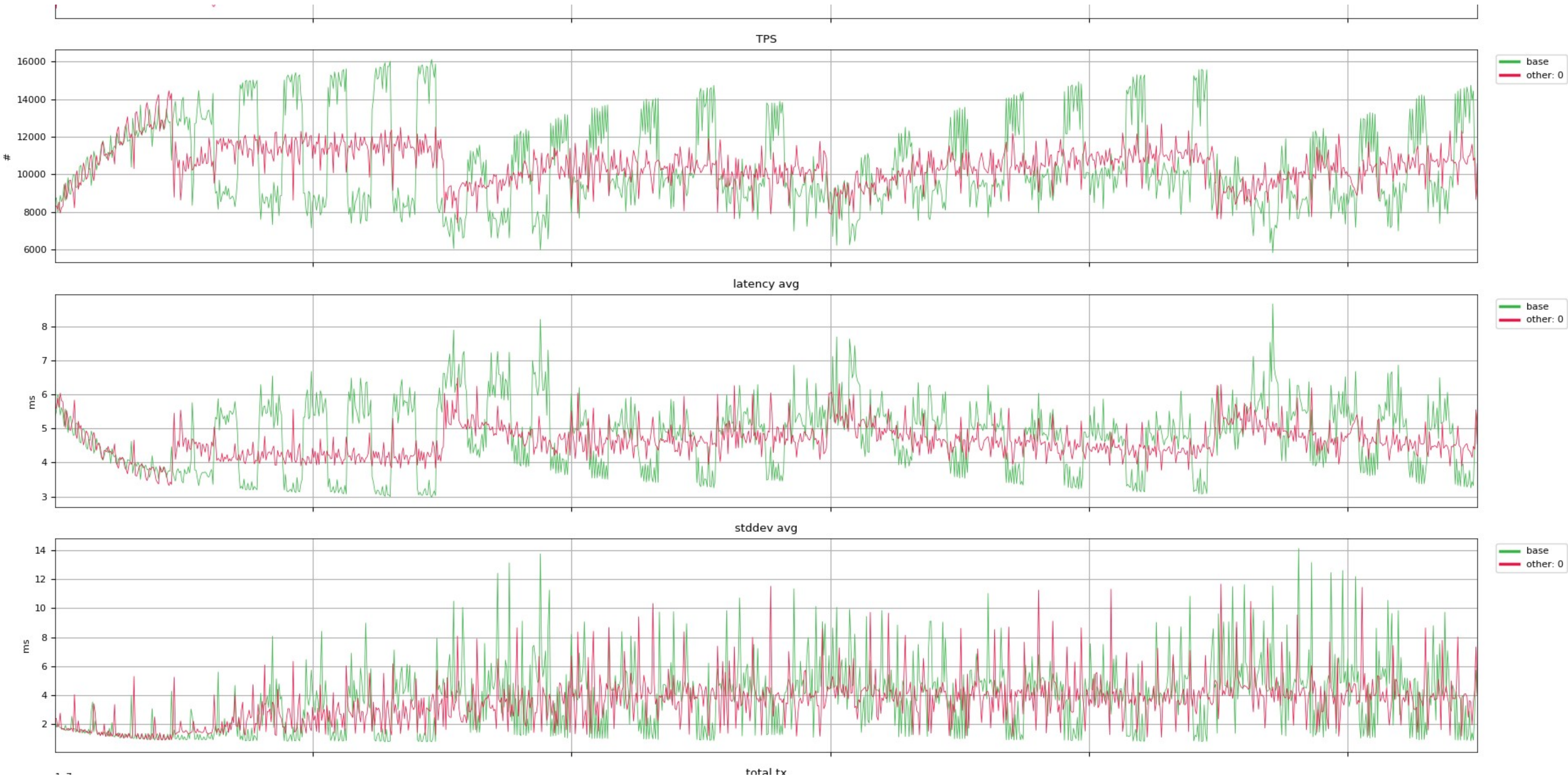
Queuing

- NVMe SSDs have enough hardware queues to have one queue per core (no locking!)
- OS level changes needed (linux: block-mq)
- IO parallelism required to benefit fully is significant
- NVMe: Each queue can be deep (thousand)
- SATA: One queue with 32 entries
- SAS / SCSI: one / few queues, with hundreds entries

Why care? Postgres uses the OS, abstracting this?

- Not utilizing hardware parallelism – not issuing enough requests in parallel
  - posix_fadvise(WILLNEED) has significant synchronous cost
- Overhead of page cache significant – and largely **synchronous**
  - synchronous scans cannot utilize hardware
- Latency **highly variable** – kernel does not have necessary information (nor interfaces to transport such information)
  - hacks with posix_fadvise(DONTNEED) make situation less bad, but not good
  - Checkpoints still have bad performance impact
  - Very hard to control better from postgres
- WAL throughput is quite low

Unpredictable Latency



Cost of memory copies from pagecache

```
Samples: 3K of event 'cycles', Event count (approx.): 1003579986
  Overhead  Command   Shared Object       Symbol
+   22.03%  postgres  [kernel.vmlinux]    [k] copy_user_enhanced_fast_string
+    6.95%  postgres  postgres            [.] hash_search_with_hash_value
+    3.58%  postgres  [kernel.vmlinux]    [k] generic_file_read_iter
+    2.98%  postgres  [kernel.vmlinux]    [k] __list_del_entry_valid
+    2.49%  postgres  postgres            [.] LWLockAcquire
+    2.21%  postgres  postgres            [.] ReadBuffer_common
+    1.99%  postgres  [kernel.vmlinux]    [k] get_page_from_freelist
+    1.92%  postgres  [kernel.vmlinux]    [k] xas_load
+    1.75%  postgres  [ext4]              [k] ext4_mpage_readpages
+    1.68%  postgres  [kernel.vmlinux]    [k] __pagevec_lru_add_fn
+    1.48%  postgres  postgres            [.] LWLockRelease
+    1.19%  postgres  [kernel.vmlinux]    [k] find_get_entry
+    1.15%  postgres  [kernel.vmlinux]    [k] __domain_mapping
+    1.15%  postgres  [kernel.vmlinux]    [k] workingset_refault
+    1.13%  postgres  [kernel.vmlinux]    [k] try_charge
+    1.03%  postgres  postgres            [.] LockBufHdr
+    0.97%  postgres  libpthread-2.29.so  [.] __libc_pread64
Tip: Generate a script for your data: perf script -g <lang>
```

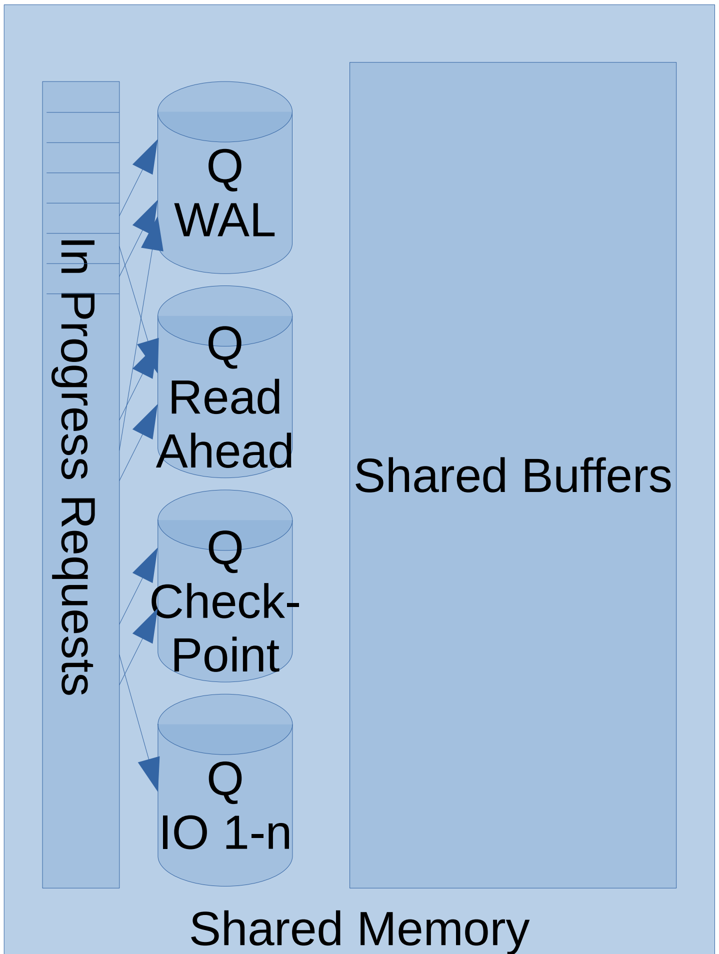
Asynchronous IO

- (often) multiple commands can be submitted at once
  - syscall overhead mitigated
- (often) DMA directly between drive and userspace memory (no kernel)
- (sometimes) commands executed via (kernel) threads

Overview of AIO APIs

- linux libaio:
  - buffered io, fsyncs fall back to synchronous execution → not suitable
  - unbuffered io: if all goes well: dma into buffers, can achieve very high speed
- windows iocp:
  - mature
  - uses threads (bad for postgres)
  - Unclear if it does DMA for unbuffered IO?
- posix aio:
  - emulated on at least some operating systems (linux)
- freebsd aio:
  - kernel threads
  - integrated with kqueue
  - Unclear if it does DMA for unbuffered IO?
- OSX
  - kernel threads,
  - apparently not integrated with kqueue (hat tip to Thomas Munro)
- linux: io uring
  - very new API (5.1, early 2019)
  - two ring buffers, very little locking
    - fewer / no syscalls in hot path
    - no locks needed
  - increasing number of operations
  - unbuffered: DMA into buffers
  - buffered: kernel threads
  - allow interdependent operations to be queued
    - e.g. start following write(s) only after prior completed

Proposed Postgres AIO Architecture



- Abstraction hiding used AIO interface
- Completion Based, AIO implementation independent callbacks (e.g. to mark async read buffer as valid)
- Multiple queues
  - WAL queue for WAL and buffer writes when dependent on WAL flush
  - Readahead queue to control maximum RA
  - Checkpoint queue
    - shallow, to control latency impact
  - Multiple IO queues for the rest
    - to achieve higher concurrency
- APIs to asynchronously read / write buffer

Comparing sync/async IO execution

synchronous read:

- allocate shared buffer
- mark buffer as IO in progress
- synchronously pread()
- mark buffer valid
- continue execution using buffer

asynchronous read:

- allocate shared buffer
- mark buffer as IO in progress
- create AIO request
- associate buffer with IO object
- (repeat)
- start multiple IOS w/ single syscall
- do something else (e.g. process previously read blocks)
- execute IO completions
- continue execution using buffer

AIO Details

- AIO implementation hidden behind generic API
  - currently API exposes high level ops like read buffer, write buffer, write wal
- Deadlock Danger:
  - p1: start reading buffer #1
  - p1: do something else, block on p2
  - p2: need buffer #1
  - Solution: p2 can complete p1’s IO, and use the buffer
- Closing File Descriptors
  - can’t re-issue requests (e.g. partial reads/writes) to shared queue from different process with same fd (number different)

Prototype

- Only supports linux's io_uring
  - but most details hidden within aio.c
- Highly experimental / unstable
- Only a single queue for now

Prototype Results

- all recent ones with linux 5.5, Samsung 970 EVO Plus 2TB
- sequential scans:
  - single process, pg prewarm:
  - buffered sync: 1.8GB/s ~75% CPU
  - unbuffered sync: 600MB/s ~20% CPU
  - buffered async: ~2GB/s, 150% CPU - too many small requests
  - unbuffered async: 3.2GB/s ~50% CPU
- parallel sequential scan 3 processes (2 workers):
  - buffered sync: 2.2 GB/s
  - unbuffered async: 3.1 GB/s
  - high latency system: not worth comparing, basically cheating, sync so bad

These benchmarks are nearly lies

Prototype Results

- larger than memory pgbench, with async writeback
  - ~20% gain, lots more to get
- WAL, open_datasync, OLTP, unbuffered (likely buggy):
  - ~15-20% gain from AIO in stupidest possible implementation
    - older version: higher gain for high latency, but definitely buggy, so ?
  - plenty to gain for *non* async too
    - split write from sync lock
    - stop writing so much at once, release waiters earlier

These benchmarks are nearly lies

Prototype Results

- WAL, open_datasync, OLTP, asynchronous commit, unbuffered (likely buggy):
  - ~30% gain
- WAL, parallel COPY of large files:
  - ~40% gain, bottleneck quickly becomes data file IO

These benchmarks are nearly lies

Subsystem Thoughts

- eventually good defaults would probably be to use unbuffered IO for writes, buffered reads (except for large seqscans, vacuum etc)
- checkpoints
  - can be sped up a good bit on busy systems, most importantly we can control latency impact (shallower queue)! Doesn't work yet in prototype
- background writer / backend writeback
  - very substantial gains by not blocking during backend writes
  - **get rid of bgwriter?**
  - Issue writes from bounce buffers?
    - very short locking duration for writes
    - memcpy not free, but already needed with checksums

Subsystem Thoughts

- Sequential Scans need own readahead logic for direct IO
  - nontrivial to compute how much to prefetch, especially on high latency systems
  - a lot more robust than using OS (random cached buffers defeat)
- FlushBuffer()
  - can issue interdependent linked IO without PG blocking
  - helps VACUUM **massively** due to ringbuffer constantly causing WAL flushes

Questions

- Do we need to support multiple platforms initially?
  - perhaps add io_uring and worker process based implementation?
  - if windows: how to deal with number of threads?
- Need to start/issue pending local requests when potentially blocking – how?
- How to efficiently wait for multiple Condition Variables?