Asynchronous IO for PostgreSQL

Andres Freund
PostgreSQL Developer & Committer
Microsoft

andres@anarazel.de
andres.freund@microsoft.com
@AndresFreundTec

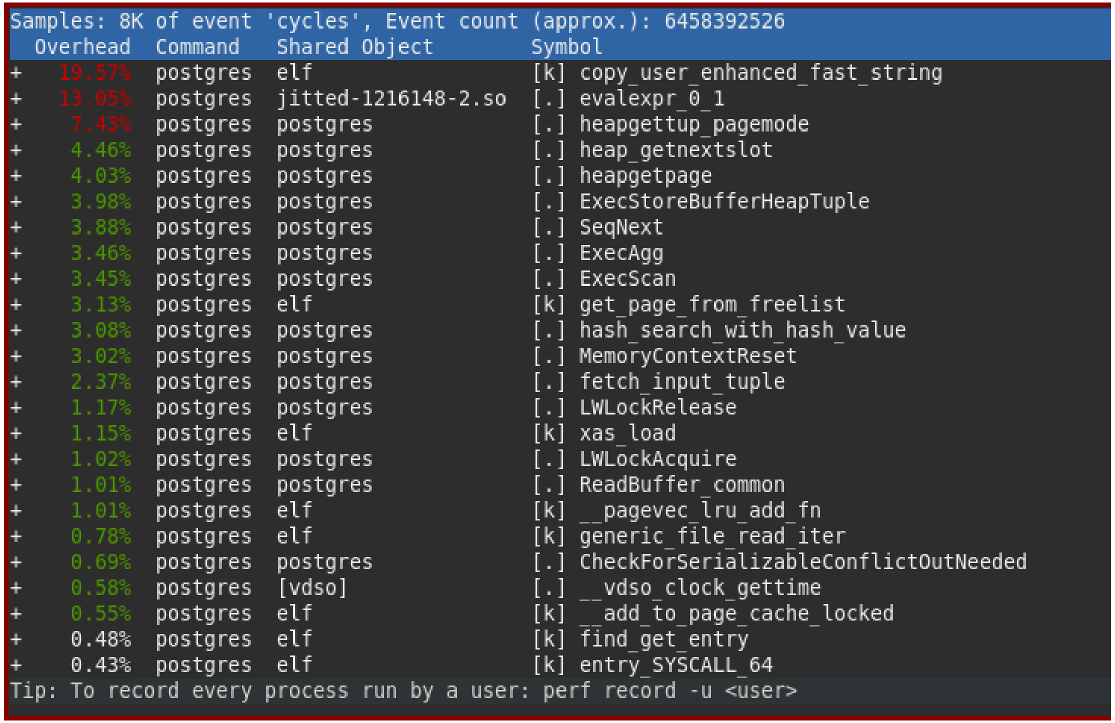
Why AIO?

Buffered IO is a major limitation.

Why AIO?

```
tpch_100[1575595][1]=# EXPLAIN (ANALYZE, BUFFERS) SELECT sum(l_quantity) FROM lineitem ;
┌────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┐
│                                                                 QUERY PLAN                                                                                 │
├────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┤
│ Finalize Aggregate  (cost=11439442.89..11439442.90 rows=1 width=8) (actual time=46264.524..46264.524 rows=1 loops=1)
│   Buffers: shared hit=2503 read=10602553
│   I/O Timings: read=294514.747
│   ->  Gather  (cost=11439441.95..11439442.86 rows=9 width=8) (actual time=46250.927..46278.690 rows=9 loops=1)
│         Workers Planned: 9
│         Workers Launched: 8
│         Buffers: shared hit=2503 read=10602553
│         I/O Timings: read=294514.747
│         ->  Partial Aggregate  (cost=11438441.95..11438441.96 rows=1 width=8) (actual time=46201.154..46201.154 rows=1 loops=9)
│               Buffers: shared hit=2503 read=10602553
│               I/O Timings: read=294514.747
│               ->  Parallel Seq Scan on lineitem  (cost=0.00..11271764.76 rows=66670876 width=8) (actual time=0.021..40497.515 rows=66670878 loops=9)│
│                     Buffers: shared hit=2503 read=10602553
│                     I/O Timings: read=294514.747
│ Planning Time: 0.139 ms
│ JIT:
│   Functions: 29
│   Options: Inlining true, Optimization true, Expressions true, Deforming true
│   Timing: Generation 5.209 ms, Inlining 550.852 ms, Optimization 266.074 ms, Emission 163.010 ms, Total 985.145 ms
│ Execution Time: 46279.595 ms
└────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┘
(20 rows)
```

Why AIO?



```
Samples: 8K of event 'cycles', Event count (approx.): 6458392526
  Overhead  Command   Shared Object         Symbol
+   19.57%  postgres  elf                   [k] copy_user_enhanced_fast_string
+   13.05%  postgres  jitted-1216148-2.so   [.] evalexpr_0_1
+    7.43%  postgres  postgres              [.] heapgettup_pagemode
+    4.46%  postgres  postgres              [.] heap_getnextslot
+    4.03%  postgres  postgres              [.] heapgetpage
+    3.98%  postgres  postgres              [.] ExecStoreBufferHeapTuple
+    3.88%  postgres  postgres              [.] SeqNext
+    3.46%  postgres  postgres              [.] ExecAgg
+    3.45%  postgres  postgres              [.] ExecScan
+    3.13%  postgres  elf                   [k] get_page_from_freelist
+    3.08%  postgres  postgres              [.] hash_search_with_hash_value
+    3.02%  postgres  postgres              [.] MemoryContextReset
+    2.37%  postgres  postgres              [.] fetch_input_tuple
+    1.17%  postgres  postgres              [.] LWLockRelease
+    1.15%  postgres  elf                   [k] xas_load
+    1.02%  postgres  postgres              [.] LWLockAcquire
+    1.01%  postgres  postgres              [.] ReadBuffer_common
+    1.01%  postgres  elf                   [k] __pagevec_lru_add_fn
+    0.78%  postgres  elf                   [k] generic_file_read_iter
+    0.69%  postgres  postgres              [.] CheckForSerializableConflictOutNeeded
+    0.58%  postgres  [vdso]                [.] __vdso_clock_gettime
+    0.55%  postgres  elf                   [k] __add_to_page_cache_locked
+    0.48%  postgres  elf                   [k] find_get_entry
+    0.43%  postgres  elf                   [k] entry_SYSCALL_64
Tip: To record every process run by a user: perf record -u <user>
```

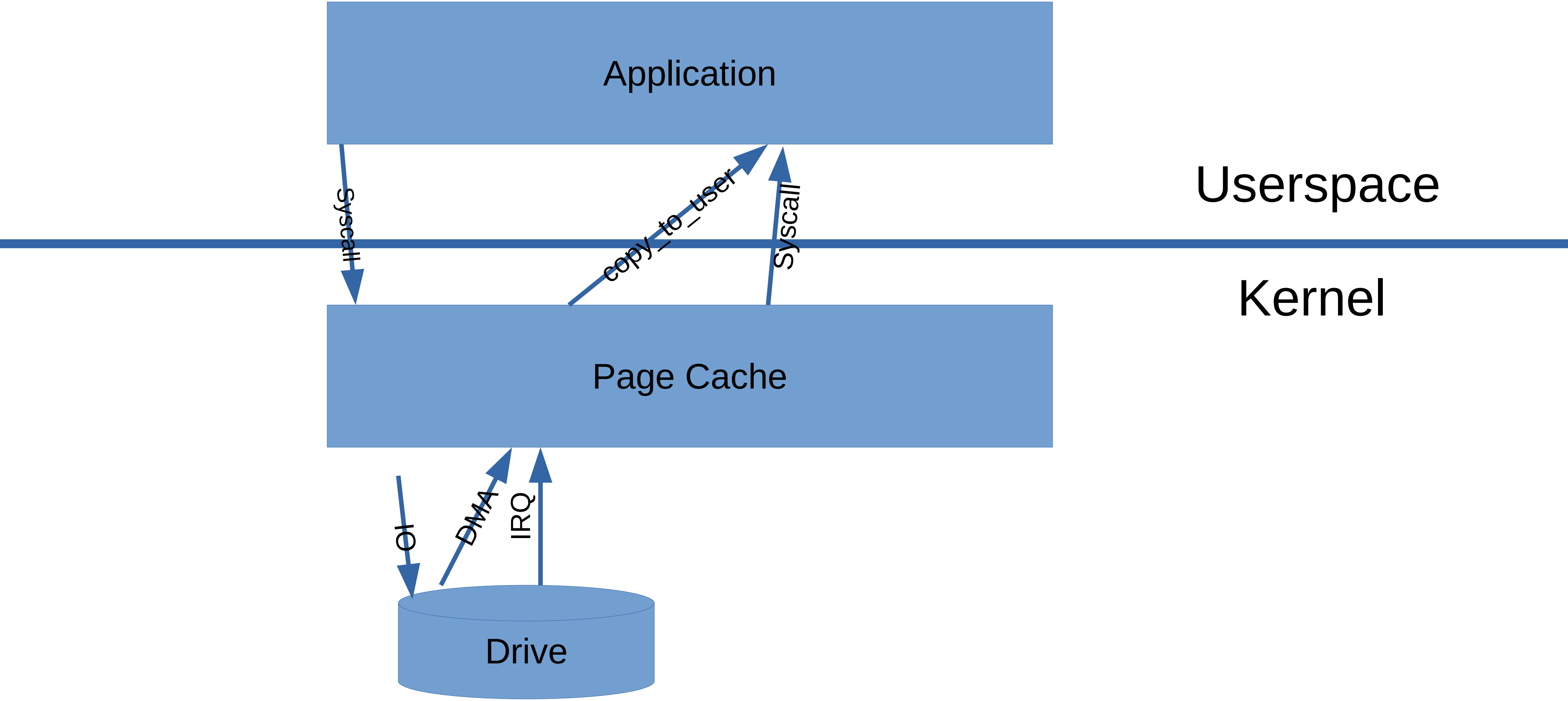
```
$ perf stat -a -e cycles:u,cycles:k,ref-cycles:u,ref-cycles:k sleep 5
 Performance counter stats for 'system wide':
    52,886,023,568      cycles:u                                                   (49.99%)
    50,676,736,054      cycles:k                                                   (74.99%)
    47,563,244,024      ref-cycles:u                                               (75.00%)
    46,187,922,930      ref-cycles:k                                               (25.00%)
       5.002662309 seconds time elapsed
```

Why AIO?

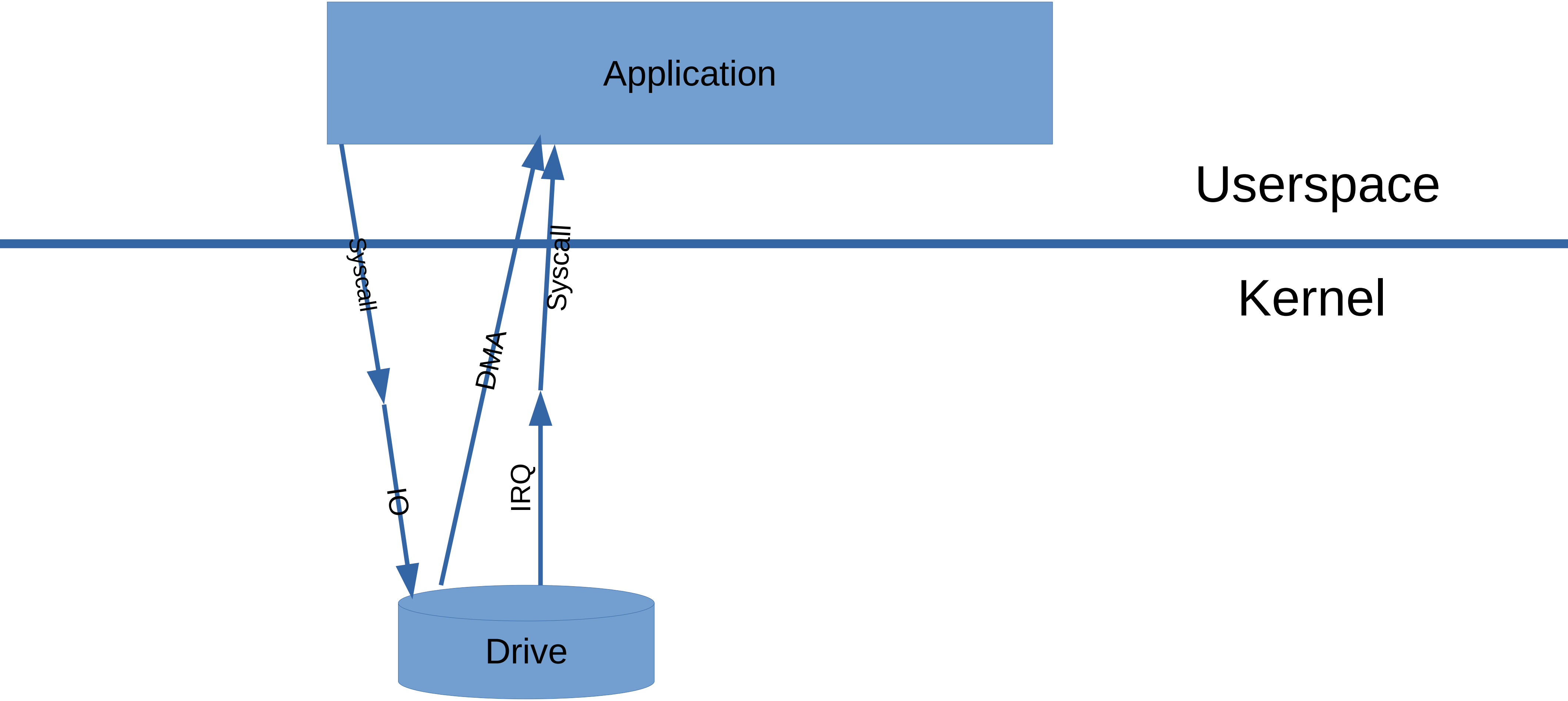
Type	Workers	Time
from disk	0	59.94s
from disk	3	48.56s
from disk	9	37.84s
from os cache	0	47.28s
from os cache	9	8.13s
from PG cache	0	34 s
from PG cache	9	5.37s

- With no amount of concurrency the disk bandwith for streaming reads (~3.2GB/s) can be reached.
- Kernel pagecache is not much faster than doing the IO for single process

Buffered uncached read()



Direct IO read()



“Direct IO”

- Kernel <-> Userspace buffer transfer, without a separate pagecache
  - Often using DMA, i.e. not using CPU cycles
  - Very little buffering in kernel
- Userspace has much much more control / responsibility when using DIO
- No readahead, no buffered writes => read(), write() are synchronous
- Synchronous use unusably slow for most purposes

Why AIO?

- Throughput problems:
  - background writer quickly saturated (leaving algorithmic issues aside)
  - checkpointer can’t keep up
  - WAL write flushes too slow / latency too high
- CPU Overhead
  - memory copy for each IO (CPU time & cache effects)
  - pagecache management
  - overhead of filesystem + pagecache lookup for small IOs
- Lack of good control
  - Frequent large latency spikes due to kernel dirty writeback management
  - Kernel readahead fails often (segment boundary, concurrent accesses, low QD for too fast / too high latency drives)
  - Our “manual” readahead comes at high costs (double shared_buffers lookup, double OS pagecache lookup, unwanted blocking when queue depth is reached, …)
- ...

Buffered vs Direct

Query	Branch	Time s	Avg CPU %	Avg MB/s
`select pg_prewarm('lineitem', ‘read’);`	master	34.6	~78	~2550
`select pg_prewarm('lineitem', 'read_aio');`	aio	27.0	~51	~3100
`select pg_prewarm('lineitem', ‘buffer’);`	master	56.6	~95	~1520
`select pg_prewarm('lineitem', 'buffer_aio');`	aio	29.3	~75	~2900

Why not yet?

- Linux AIO didn’t use to support buffered IO
- Not everyone can use DIO
- Synchronous DIO very slow (latency)

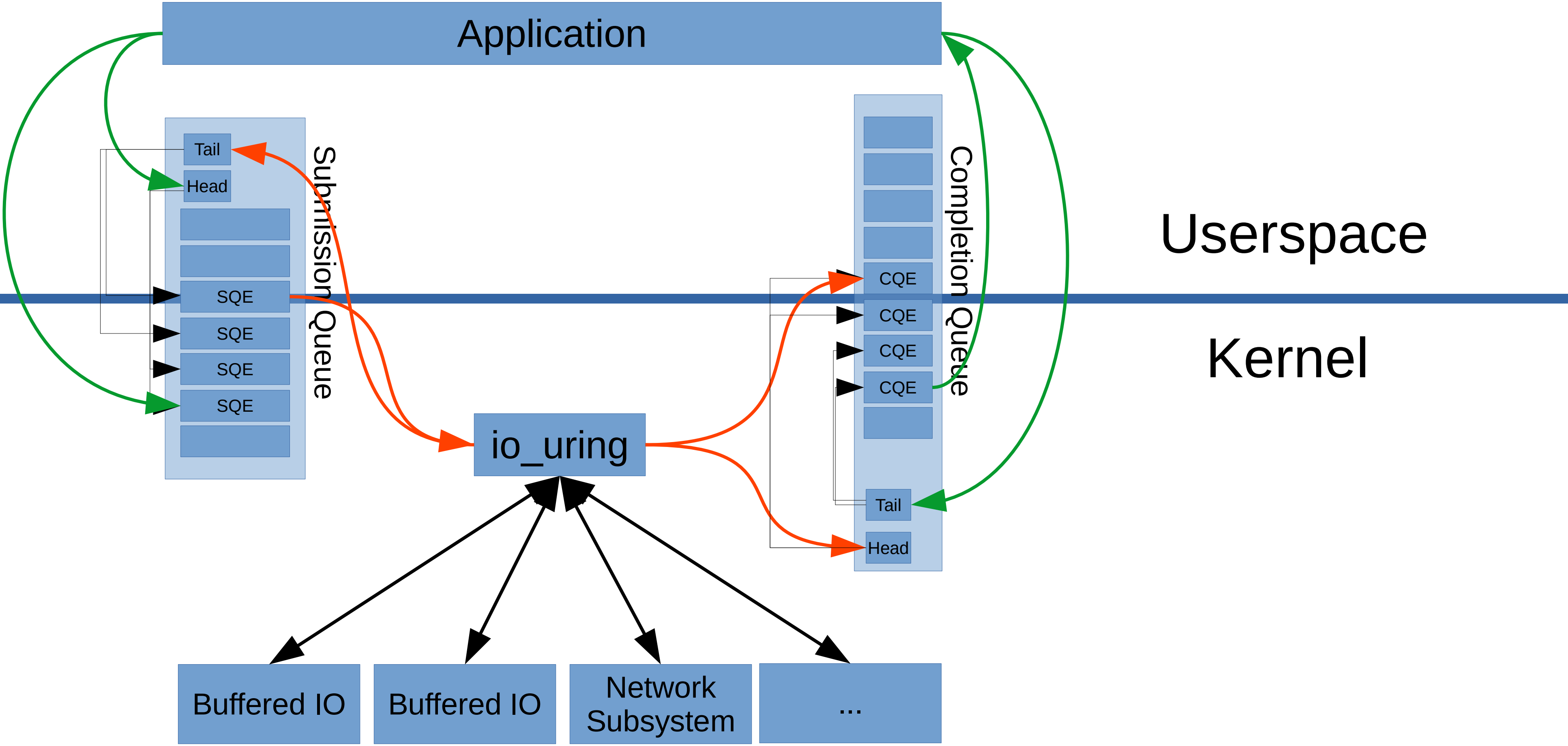
- It’s a large project / most people are sane
- Adds / Requires complexity

```
postgres[1583389][1]=# SHOW io_data_direct ;
┌────────────────┐
│ io_data_direct │
├────────────────┤
│ on             │
└────────────────┘
tpch_100[1583290][1]=# select pg_prewarm('lineitem', 'read');
┌────────────┐
│ pg_prewarm │
├────────────┤
│   10605056 │
└────────────┘
(1 row)

Time: 160227.904 ms (02:40.228)


Device       r/s        rMB/s     rrqm/s   %rrqm  r_await  rareq-sz    aqu-sz %util
nvme1n1      71070.00   555.23    0.00     0.00   0.01     8.00        0.00   100.00
```

io_uring

- New linux AIO interface, added in 5.1
- Generic, quite a few operations supported
  - open / close / readv / writev / fsync, statx, …
  - send/recv/accept/connect/…, including polling
- One single-reader / single writer ring for IO submission, one SPSC ring for completion
  - allows batched “syscalls”
- Operations that aren’t fully asynchronous are made asynchronous via kernel threads

io_uring basics



io_uring operations

```
/*
 * IO submission data structure (Submission Queue Entry)
 */
struct io_uring_sqe {
        __u8    opcode;         /* type of operation for this sqe */
        __u8    flags;          /* IOSQE_ flags */
        __u16   ioprio;         /* ioprio for the request */
        __s32   fd;             /* file descriptor to do IO on */
        union {
                __u64   off;    /* offset into file */
                __u64   addr2;
        };
        union {
                __u64   addr;   /* pointer to buffer or iovecs */
                __u64   splice_off_in;
        };
        __u32   len;            /* buffer size or number of iovecs */
        union {
                __kernel_rwf_t  rw_flags;
                __u32           fsync_flags;
                __u16           poll_events;
                __u32           sync_range_flags;
                ...
        };
        __u64   user_data;      /* data to be passed back at completion
time */
```

```
enum {
        IORING_OP_NOP,
        IORING_OP_READV,
        IORING_OP_WRITEV,
        IORING_OP_FSYNC,
        IORING_OP_READ_FIXED,
        IORING_OP_WRITE_FIXED,
        IORING_OP_POLL_ADD,
        IORING_OP_POLL_REMOVE,
        IORING_OP_SYNC_FILE_RANGE,
        IORING_OP_SENDMSG,
        IORING_OP_RECVMSG,
        IORING_OP_TIMEOUT,
        IORING_OP_TIMEOUT_REMOVE,
        IORING_OP_ACCEPT,
        IORING_OP_ASYNC_CANCEL,
        IORING_OP_LINK_TIMEOUT,
        IORING_OP_CONNECT,
        IORING_OP_FALLOCATE,
        IORING_OP_OPENAT,
        IORING_OP_CLOSE,
        IORING_OP_FILES_UPDATE,
        IORING_OP_STATX,
...

        /* this goes last, obviously */
        IORING_OP_LAST,
};
```

Constraints on AIO for PG

- Buffered IO needs to continue to be feasible
- Platform specific implementation details need to be abstracted
- Cross process AIO completions are needed:
  1) backend a: lock database object x exclusively
  2) backend b: submit read for block y
  3) backend b: submit read for block z
  4) backend a: try to access block y, IO_IN_PROGRESS causes wait
  5) backend b: try to lock database object x

Lower-Level AIO Interface

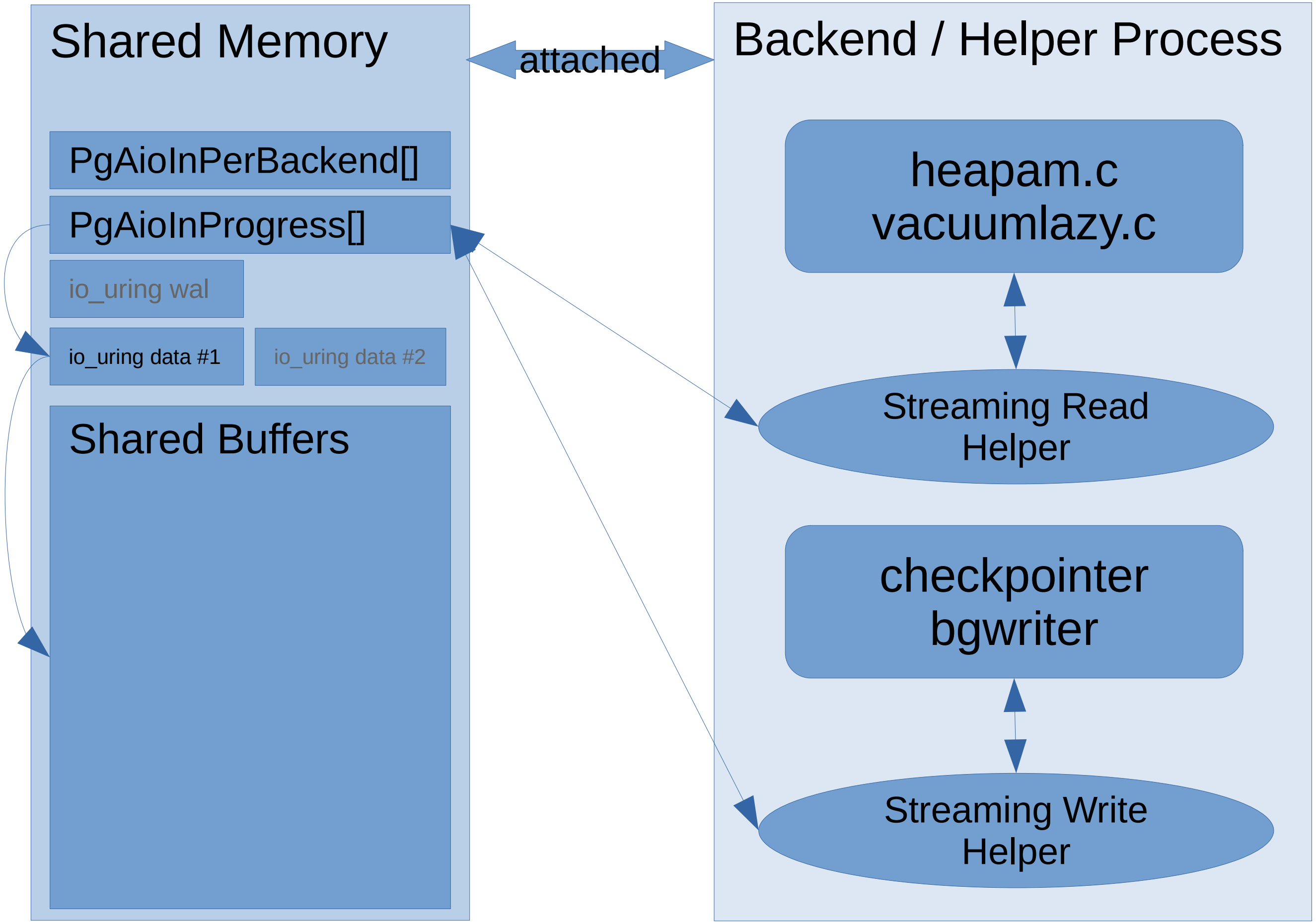
1) Acquire a shared “IO” handle
   `aio = pgaio_io_get();`
2) Optionally register callback to be called when IO completes
   `pgaio_io_on_completion_local(aio, prefetch_more_and_other_things)`
3) Stage some form of IO locally:
   `pgaio_io_start_read_buffer(aio, …)`
   a) Go back to 1) many times if useful
4) Cause pending IO to be submitted
   a) By waiting for an individual IO:
      `pgaio_io_wait()`
   b) By explicitly issuing individual IO:
      `pgaio_submit_pending()`
5) aio.c submits IO via io_uring

Higher Level AIO Interface

- Streaming Read helper
  - Tries to maintain N requests in flight, up to a certain distance from current point
  - Caller users `pg_streaming_read_get_next(pgsr);` to get the next block
  - Uses provided callback to inquire which IO is the next needed
    - heapam fetches sequentially
    - vacuum checks VM which is next
  - Uses `pgaio_io_on_completion_local()` callback to promptly issue new IOs

- Streaming Write
  - Controls the number of outstanding writes
  - allows to wait for all pending IOs (at end, or before a potentially blocking action)

Prototype Architecture



Shared Memory

attached

Backend / Helper Process

PgAioInPerBackend[]

PgAioInProgress[]

io_uring wal

io_uring data #1

io_uring data #2

Shared Buffers

heapam.c
vacuumlazy.c

Streaming Read
Helper

checkpointer
bgwriter

Streaming Write
Helper

Prototype Results

- Helps most with very high throughput low latency drives **and** with high latency & high throughput
- analytics style queries:
  - often considerably faster (TPCH 100 has all faster, several > 2x)
  - highly parallel bulk reads scale poorly, known cause (one io_uring + locks)
  - seqscan ringbuffer + hot pruning can cause problems: Ring buffers don’t use streaming write yet
- OLTP style reads/writes: A good bit better, to a bit slower
  - WAL AIO needs work
  - Better prefetching: See earlier talk by Thomas Munro
- VACUUM:
  - Much faster heap scan (~2x on low lat, >5x on high lat high throughput)
  - DIO noticably slower for e.g. btree index scans: readahead helper not yet used, but trivial
  - Sometimes slower when creating a lot of dirty pages:
- Checkpointer: >2x
- Bgwriter: >3x

Next Big Things

- Use AIO helpers in more places
  - index vacuums
  - non-bufmgr page replacement
  - better use in bitmap heap scans
  - COPY & VACUUM streaming writes
- Scalability improvements (actually use more than one io_uring)
- Efficient AIO use in WAL
- Evaluate if process based fallback is feasible?

Resources

- git tree
  - https://github.com/anarazel/postgres/tree/aio
  - https://git.postgresql.org/gitweb/?p=users/andresfreund/postgres.git;a=shortlog;h=refs/heads/aio
- Earlier talks related to AIO in PG
  - https://anarazel.de/talks/2020-01-31-fosdem-aio/aio.pdf
  - https://anarazel.de/talks/2019-10-16-pgconf-milan-io/io.pdf
- io_uring
  - “design” document: https://kernel.dk/io_uring.pdf
  - LWN articles:
    - https://lwn.net/Articles/776703/
    - https://lwn.net/Articles/810414/
  - man pages:
    - https://manpages.debian.org/unstable/liburing-dev/io_uring_setup.2.en.html
    - https://manpages.debian.org/unstable/liburing-dev/io_uring_enter.2.en.html