A fine-grained concurrent ring buffer mode for IO_CACHE

1. Rationale.

Current implementation of IO_CACHE's *`SEQ_READ_APPEND`* mode behaves coarsely grained on its write buffer: every read and every write is protected by `append_buffer_lock`.

```
int _my_b_seq_read(IO_CACHE
*info, ...) {
  lock_append_buffer(info);
  ... // read logic

  unlock_append_buffer(info);
  return Count ? 1 : 0;
}
```

```
int my_b_append(IO_CACHE *info,
...) {
  lock_append_buffer(info);
  ... // append logic

  unlock_append_buffer(info);
  return 0;
}
```

Despite the separate read buffer is read-only, and therefore is accessed wait-free, the write buffer can have a contention with medium-sized transactions.

The design described hereafter is going to solve this issue, and an extension for a parallel multi-producer workflow is additionally provided.

Furthermore, the API extension for multi-producer approach support is proposed, and the multi-consumerness is discussed.

2. The single-producer, single-consumer case.

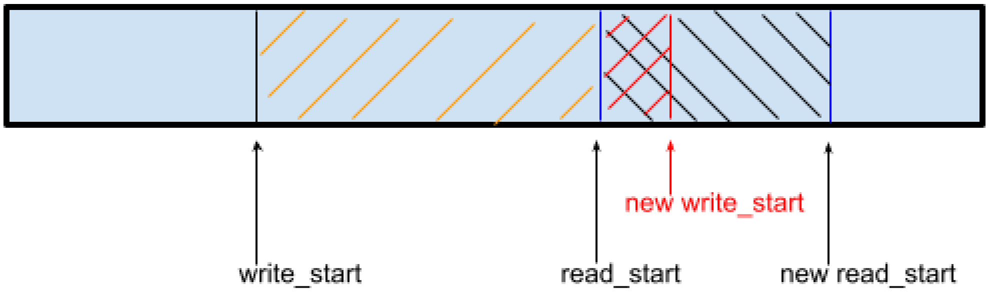
Idea

The memcpy operations of consumer and producer never overlap, therefore they can be freed of locks.

Overflow and emptiness

We cannot begin writing in the area still involved in reading. Therefore, the reader should not update the pointers before it finishes reading. This means that we should lock in the beginning to atomically read the data, and in the end, to write the new reader data.

Same for the vice-versa, we cannot read from the area still involved into writing, therefore a read should finish with `EMPTY` error (currently `_my_b_seq_read` just returns 1)



When we reach a "buffer is full" condition, we can **flip the read and write (append) buffers,** if we were reading from an append buffer. Otherwise, the append buffer is flushed.

The algorithm

The following pseudocode will describe the single-consumer, single-producer approach.
It is assumed that reading from the read buffer is handled in the usual way.

`io->total_size` and `io->read_buffer` are considered to be accessed atomically.

```
io_err_t read(IO_CACHE *io, uchar *buffer, size_t sz)
{
        if (sz > io->total_size)
                return E_IO_EMPTY;

        uchar *read_buffer = io->read_buffer;

        if (io->read_pos points to read_buffer)
                sz_read = read_from_read_buffer(io, buffer, sz);
        buffer += sz_read;
        sz -= sz_read;

        io->total_size -= sz_read;

        if (sz == 0)
                return 0;
        // else copy from append buffer

        lock(io->append_buffer_lock);
        // copy the local variables
        uchar *read_pos = io->read_pos;
        uchar *read_buffer = io->read_buffer;
        uchar *append_start_pos = i->append_start_pos;
        uchar *append_size = io->append_size;
```

```
    uchar *append_pos = io->append_pos;
    // etc, if needed
    unlock(io->append_buffer_lock);

    read from append buffer;

    lock(io->append_buffer_lock);
    // update the variables
    io->append_size -= sz;
    io->append_start_pos += sz;
    if (i->append_start_pos >= io->append_buf + io->cache_size)
        io->append_start_pos -= io->cache_size;
    unlock(io->append_buffer_lock);

    io->total_size -= sz;
}
```

The first read()'s part tries to read from a read-only buffer. If it's empty, it moves the effort to a volatile append buffer. All the metadata is copied in the first critical section, before the data copying, to the stack. It is updated back in the second critical section, after the data copying.

```
io_err_t write(IO_CACHE *io, uchar *buffer, size_t sz)
{
    lock(io->append_buffer_lock);
    if (append_buffer is full and io->total_size <= io->append_size)
        swap(io->append_buffer, io->read_buffer);
    else flush the append buffer if needed;
    write to disk directly, if the data is too large;

    uchar *write_pos = io->write_pos;
    unlock(io->append_buffer_lock);

    write to append buffer;

    lock(io->append_buffer_lock);
    io->write_pos = new_write_pos;
    unlock(io->append_buffer_lock);

    io->total_size += sz;
}
```

The important note here is that we access io->read_buffer in the reader's thread without the lock (the accesses are marked bold). However this access happens only once in the beginning and is safe:

1. Only writer changes read_buffer
2. The writer can change it only once during one read()
3. if `io->read_buffer` is considered reads-only, then it will not flip again, and continue to be consistent, until `io->total_size` is changed:

```
        io->total_size -= sz_read;
```

Then the lock happens. It should be fine to read from a flipped buffer on that stage.

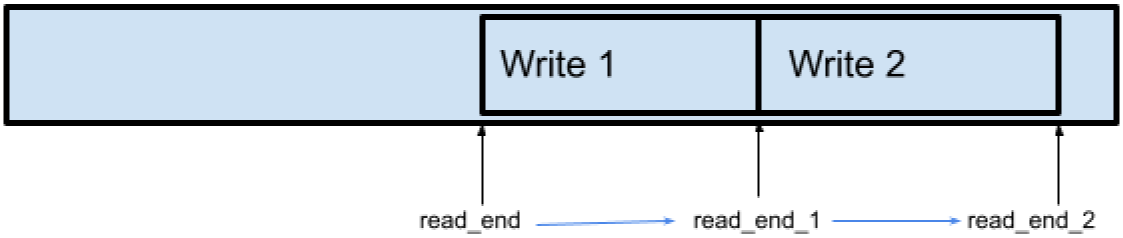
3. Multi-producer concurrency

Idea

Writes start from io->write_start, which is to update immediately. Reads are possible only until io->read_end, which is updated, as soon as writes are finished.

Medium-grained approach

io->write_start is updated immediately to allow parallel writes. However, we cannot update io->read_end immediately after this thread's write ends, because earlier writes can still be in progress. We should wait for them i.e. we wait while (io->read_end != local_read_end)



read_end cannot be updated to read_end_2 before it is updated to read_end_1

Algorithm (medium-grained)

Medium-grained approach will modify write() function as follows (the changed lines and locks are bolded):

```
io_err_t write(IO_CACHE *io, uchar *buffer, size_t sz)
{
        lock(io->append_buffer_lock);
        if (buffer flip of flush is needed)
                wait until all the writes are finished;

        if (append_buffer is full &&
            io->write_total_size <= io->append_size)
                swap(io->append_buffer, io->read_buffer);
        else flush the append buffer if needed;
        write to disk directly, if the data is too large;

        uchar *local_write_start = io->write_start;
        io->write_total_size += sz;
```

```
        io->write_start += sz;
        if (io->write_start > io->append_buffer + io->cache_size)
                io->write_start -= io->cache_size;
        unlock(io->append_buffer_lock);

        write to append buffer;

        lock(io->write_event_lock)

        while(local_write_start != io->read_end)
                cond_wait(io->write_event, io->write_event_lock);

        unlock(io->write_event_lock)



        lock(io->append_buffer_lock);
        io->read_end = new_read_end;
        unlock(io->append_buffer_lock);

        cond_signal(io->write_event);

        io->total_size += sz;
}
```

`The read function should be modified mostly cosmetically.`

Fine graining

The writers are still waiting for each other's finish. The approach described here defers waiting through **helping pattern** by introducing **progress slots**.

Each time a writer begins progress it allocates a slot in the dedicated (fixed size) array.
When the writer finishes its job, it checks whether it is the leftmost one (relative to its read_end value. If it is, it updates read_end for itself, and for all the consecutive writers already finished.

The slot allocation will be controlled by a semaphore to prevent overflow. Therefore, only a fixed number of producers can work simultaneously.

The slot array is made of elements of private cache_slot_t structure:

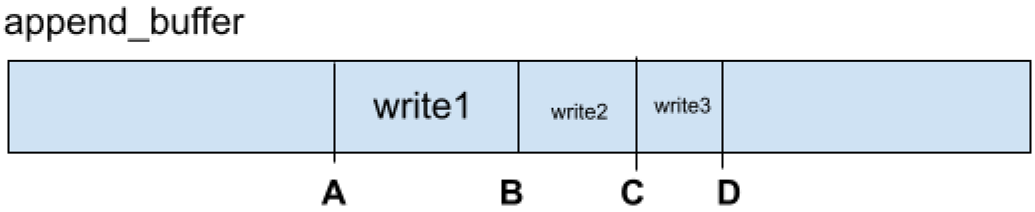
```
struct cache_slot_t {
  bool vacant: 1;
  bool finished: 1;
  uint next: size_bits(uint) - 2;
  uint pos;
```

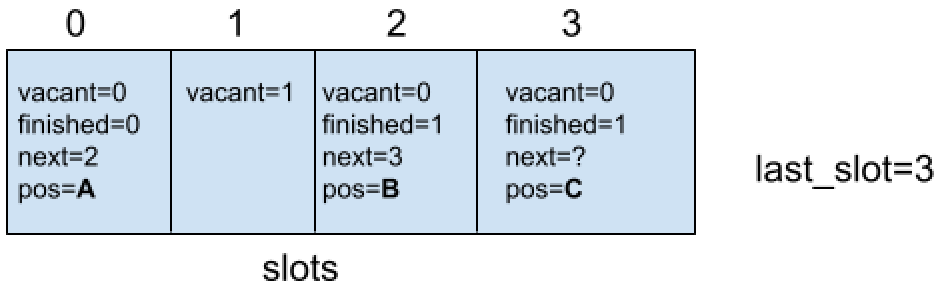
```
}
```

The slot is acquired whenever a write begins by searching an array cell with vacant=1. When it’s found, `vacant = 0, finished = 0` is set. The `last_slot` variable holds the slot index for the latest write. `slots[last_slot].next` is set to a new index, and `last_slot` itself is updated.

The following example demonstrates how the slots work: there were three writes currently running in parallel. write2 and write3 are finished, but write1 is still running. When it finishes, it will hop through slot.next while vacant==0 and finished==1 and pos != io->write_start. Therefore, read_end will be updated to C if no other write will begin in parallel.

If another write begins in parallel before write1 finishes, it allocates slots[1] and sets pos=D. slots[3].next would be set to 1, and last_slot will be updated from 3 to 1.





The slot run through expected complexity is O(1). The proof for acquisition is however not that obvious to prove the same, and no effort was spent for proving it (It’s only obvious that it’s O(slots)).

4. Arbitrary data sources support

The widely spread use-case is pouring from another IO_CACHE source (like a statement or transaction cache). The operation may require several consecutive write() calls with an external lock:

```
lock(write_lock);
uchar buffer[SIZE];
while(cache_out is not empty) {
```

```
  read(ceche_out, buffer, SIZE);
  write(cache_in, buffer, SIZE);
}
unlock(write_lock);
```

This case destroys all the parallel design described.
However, let’s make api changes to allow blocks of predicted size be written in parallel:

```
/** Allocates the slot of a requested size for a writer. Returns new slot id. */
slot_id_t append_allocate(IO_CAHCE*, size_t block_size);

/** Frees the slot and propagates the data to be available for reading */
viod append_commit(IO_CACHE*, slot_id_t);
```

These two functions just decompose our write() function: append_allocate would include the first critical section and append_commit would include the second one.

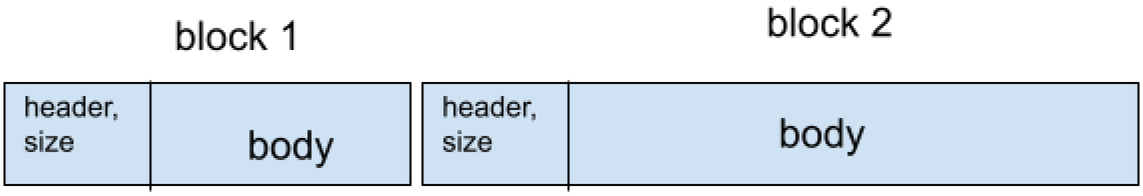
The use-case will be changed slightly:

```
slot_id_t slot = append_allocate(cache_out, append_tell(cache_in));
uchar buffer[SIZE];
while(cache_out is not empty) {
  read(ceche_out, buffer, SIZE);
  write(cache_in, buffer, SIZE);
}
append_commit(cache_out, slot);
```

5. Multi-consumerness

We currently have no cases with several readers working in parallel in SEQ_READ_APPEND mode. It is only used by the replication thread to read out the log, where it is delegated to a dedicated worker. The first problem is that parallel readout would require additional coordination -- the order of event application can be important.

Another problem is that a variable-sized blocks require at least two consecutive reads if the structure is not known. If the length is stored, it can be read out with exactly two reads (first reads length, second reads the body).



The slot allocation strategy can be applied, and api can be added similar to a new write api:

```
/** lock the cache and allocate the read slot */
slot_id_t read_allocate_lock(IO_CACHE*);
```

```
/** Allocate a read zone of the requested size and unlock the cache */
void read_allocate_unlock(IO_CACHE*, slot_id_t, size_t size);
/** Finish reading; deallocate the read slot */
void read_commit(IO_CACHE*, slot_id_t);
```

Reading api needs one function more than writing api -- the allocation is split on two phases: locking phase (to compute the block length), and the actual requesting phase.

This approach has several disadvantages:

1. The read buffer access is no longer lock-free
2. `read_allocate_lock` leaves the IO_CACHE in a locked state, which can be potentially misused.

Additionally, two SX locks can be used (one for readers and one for writers) for extra parallelism.