MySQL Buffer Management

Mijin Ahn

meeeeejin@gmail.com

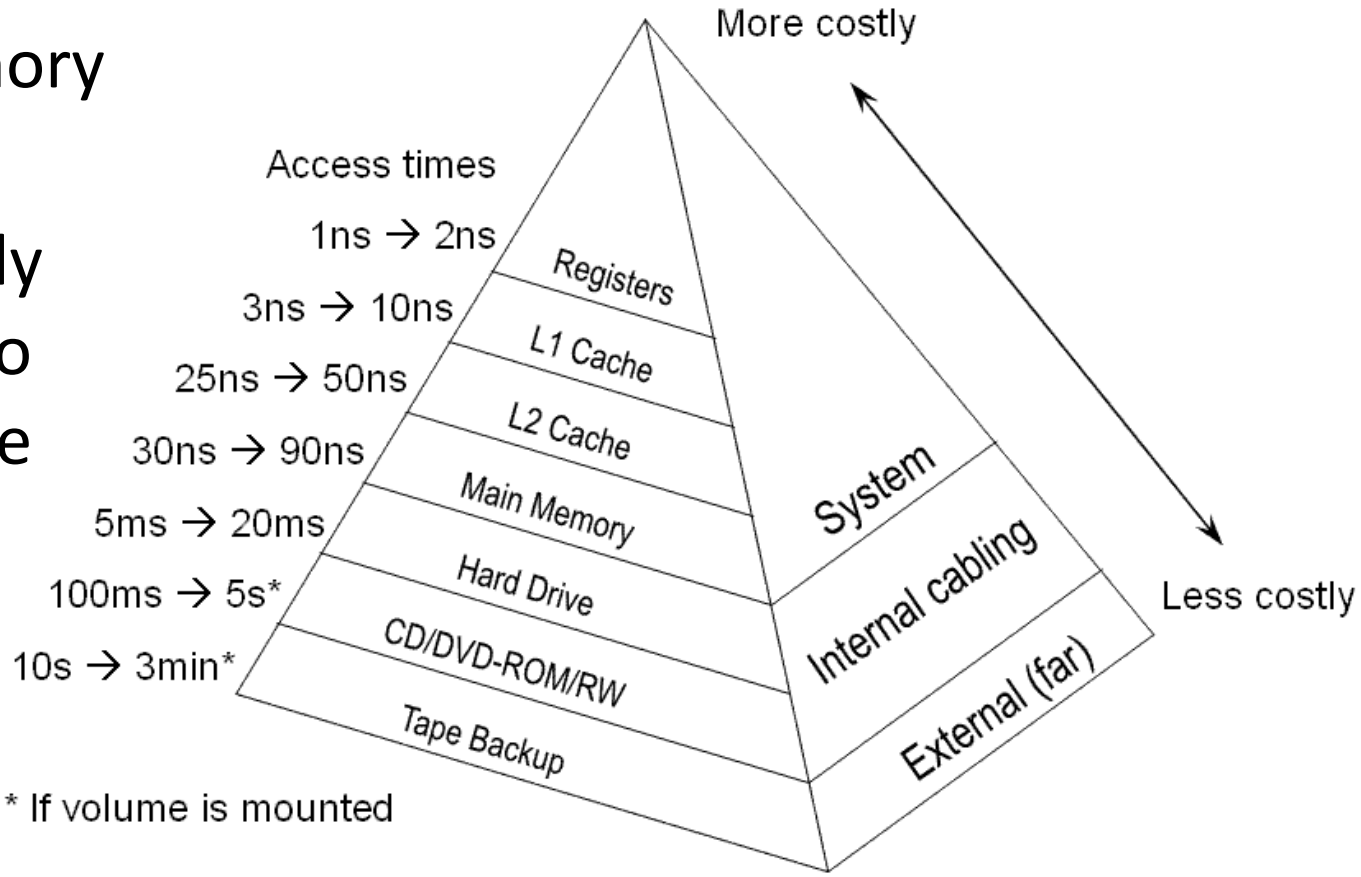
Contents

- Overview
- Buffer Pool
- Buffer Read
- LRU Replacement
- Flusher
- Doublewrite Buffer
- Synchronization

OVERVIEW

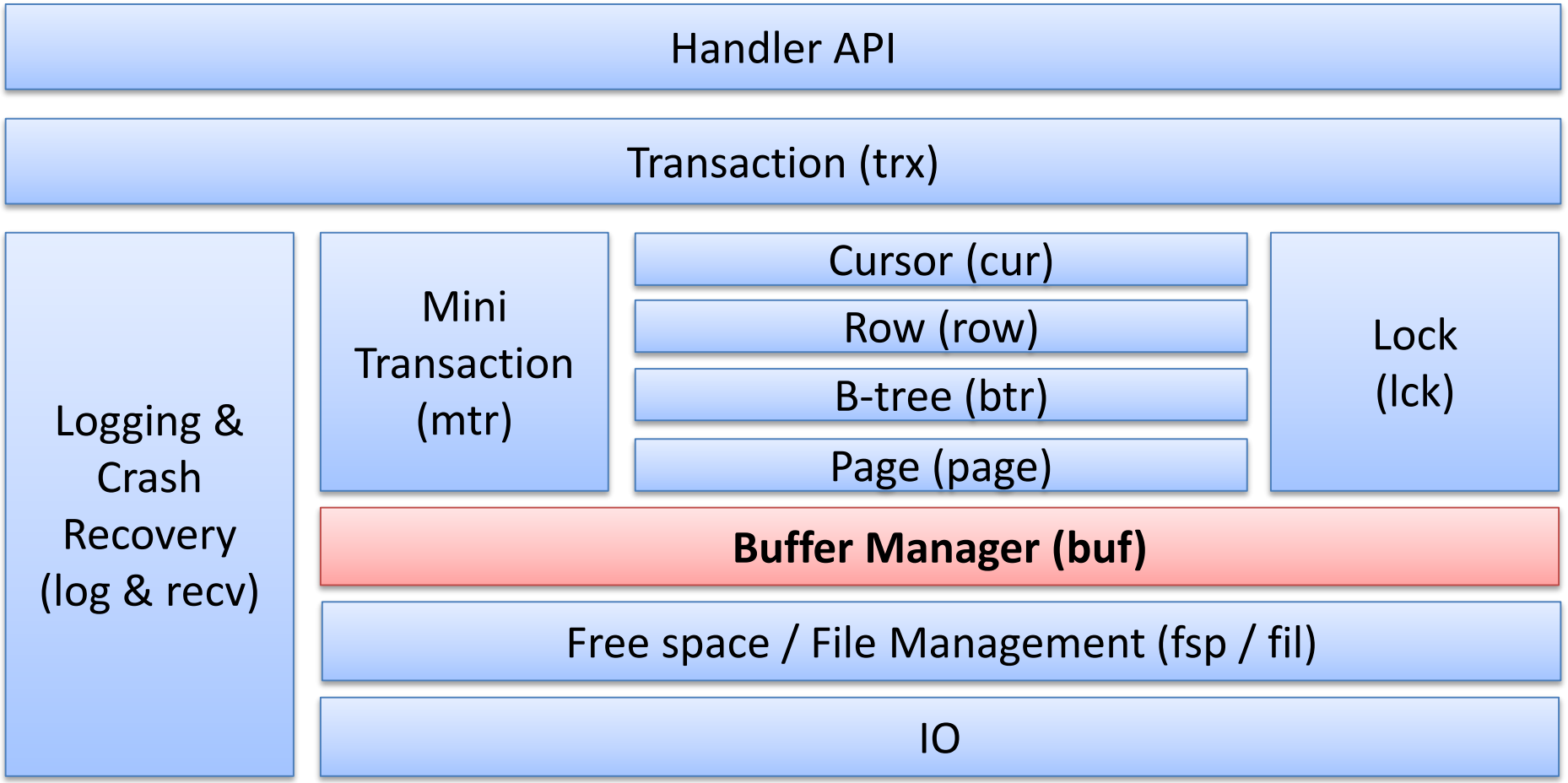
Overview

- Buffer Pool
  - Considering memory hierarchy
  - Caching frequently accessed data into DRAM like a cache memory in CPU
  - Exploit locality



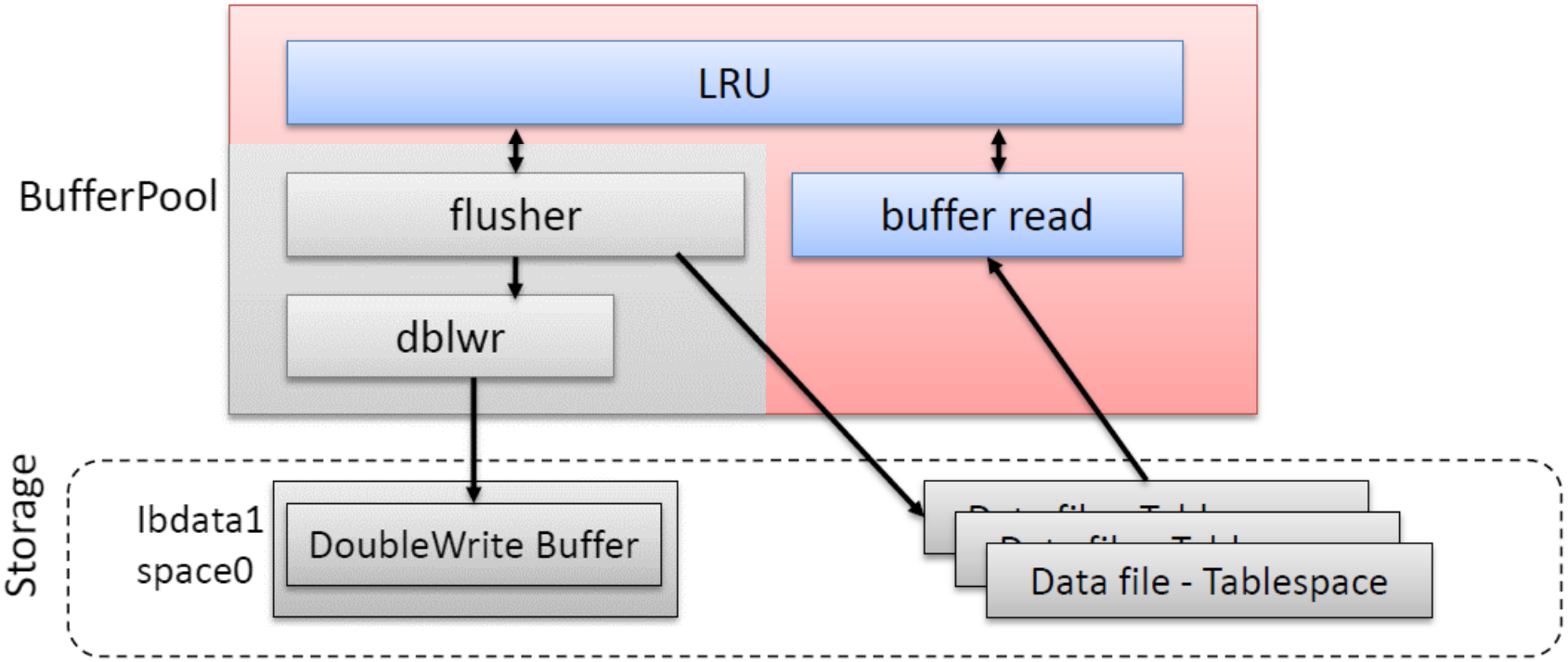
Overview

- InnoDB Architecture



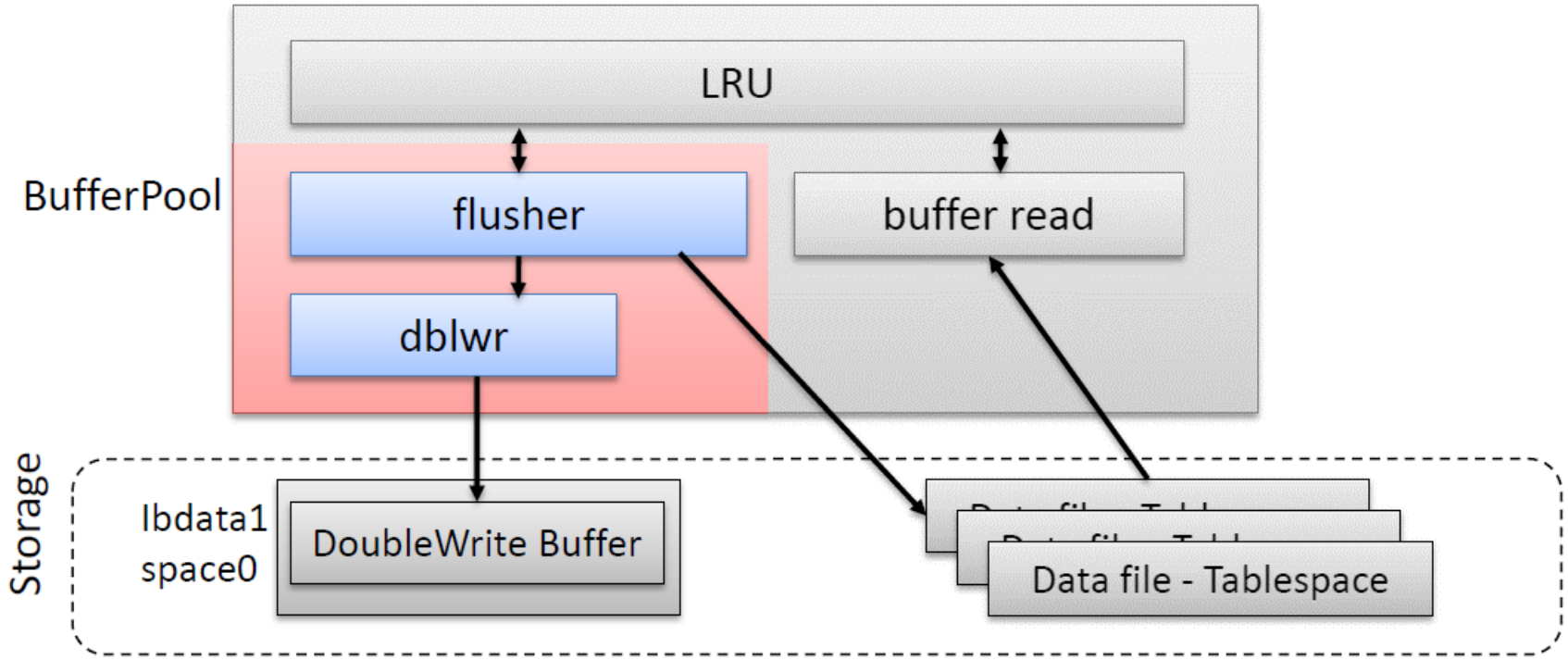
Overview

- Buffer Manager
  - Buffer Pool (buf0buf.cc) : buffer pool manager
  - Buffer Read (buf0read.cc) : read buffer
  - LRU (buf0lru.cc) : buffer replacement



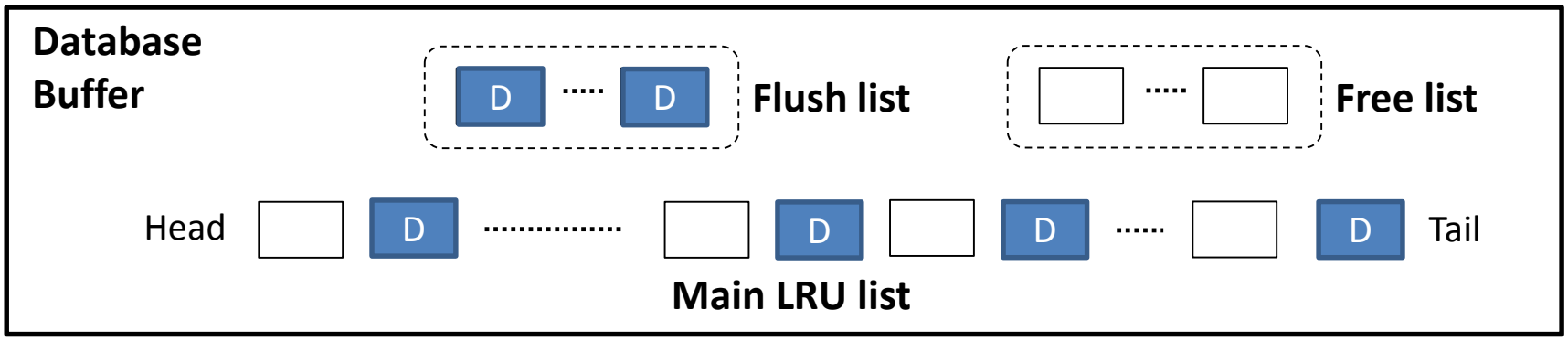
Overview

- Buffer Manager
  - Flusher (buf0flu.cc) : dirty page writer & background flusher
  - Doublewrite (buf0dblwr.cc) : doublewrite buffer



BUFFER POOL

Lists of Buffer Blocks



- **Free list**
  - Contains free page frames
- **LRU list**
  - Contains all the blocks holding a file page
- **Flush list**
  - Contains the blocks holding file pages that have been modified in the memory but not written to disk yet

Buffer Pool Mutex

- The buffer *buf_pool* contains a single mutex
  - *buf_pool->mutex*: protects all the control data structures of the *buf_pool*

- The *buf_pool->mutex* is a **hot-spot** in main memory
  - Causing a lot of memory bus traffic on multiprocessor systems when processors alternatively access the mutex
  - A solution to reduce mutex contention
    - To create a separate lock for the page hash table

Buffer Pool Struct

- include/buf0buf.h: buf_pool_t

```
1830  struct buf_pool_t{
1831
1832          /** @name General fields */
1833          /* @{ */
1834          ib_mutex_t      mutex;          /*!< Buffer pool mutex of this
1835                                          instance */
1836          ib_mutex_t      zip_mutex;      /*!< Zip mutex of this buffer
1837                                          pool instance, protects compressed
1838                                          only pages (of type buf_page_t, not
1839                                          buf_block_t */
1840          ulint           instance_no;    /*!< Array index of this buffer
1841                                          pool instance */
1842          ulint           old_pool_size;  /*!< Old pool size in bytes */
1843          ulint           curr_pool_size; /*!< Current pool size in bytes */
1844          ulint           LRU_old_ratio;  /*!< Reserve this much of the buffer
1845                                          pool for "old" blocks */
```

Buffer Pool Struct

- include/buf0buf.h: buf_pool_t

```
1853        ulint              n_chunks;       /*!< number of buffer pool chunks */
1854        buf_chunk_t*       chunks;         /*!< buffer pool chunks */
1855        ulint              curr_size;      /*!< current pool size in pages */
1856        hash_table_t*      page_hash;      /*!< hash table of buf_page_t or
1857                                              buf_block_t file pages,
1858                                              buf_page_in_file() == TRUE,
1859                                              indexed by (space_id, offset).
1860                                              page_hash is protected by an
1861                                              array of mutexes.
1862                                              Changes in page_hash are protected
1863                                              by buf_pool->mutex and the relevant
1864                                              page_hash mutex. Lookups can happen
1865                                              while holding the buf_pool->mutex or
1866                                              the relevant page_hash mutex. */
1867        hash_table_t*      zip_hash;       /*!< hash table of buf_block_t blocks
1868                                              whose frames are allocated to the
1869                                              zip buddy system,
1870                                              indexed by block->frame */
```

Buffer Pool Struct

- include/buf0buf.h: buf_pool_t

```
1890        ib_mutex_t      flush_list_mutex;/*!< mutex protecting the
1891                                         flush list access. This mutex
1892                                         protects flush_list, flush_rbt
1893                                         and bpage::list pointers when
1894                                         the bpage is on flush_list. It
1895                                         also protects writes to
1896                                         bpage::oldest_modification and
1897                                         flush_list_hp */
1898        const buf_page_t*       flush_list_hp;/*!< "hazard pointer"
1899                                         used during scan of flush_list
1900                                         while doing flush list batch.
1901                                         Protected by flush_list_mutex */
1902        UT_LIST_BASE_NODE_T(buf_page_t) flush_list;
1903                                         /*!< base node of the modified block
1904                                         list */
1905        ibool           init_flush[BUF_FLUSH_N_TYPES];
1906                                         /*!< this is TRUE when a flush of the
1907                                         given type is being initialized */
```

Buffer Pool Struct

- include/buf0buf.h: buf_pool_t

```
1948        /* @} */
1949
1950        /** @name LRU replacement algorithm fields */
1951        /* @{ */
1952
1953        UT_LIST_BASE_NODE_T(buf_page_t) free;
1954                                        /*!< base node of the free
1955                                        block list */
1956        UT_LIST_BASE_NODE_T(buf_page_t) LRU;
1957                                        /*!< base node of the LRU list */
```

Buffer Pool Init

- buf/buf0buf.cc: buf_pool_init()

```
1486  buf_pool_init(
1487  /*==========*/
1488          ulint   total_size,     /*!< in: size of the total pool in bytes */
1489          ulint   n_instances)    /*!< in: number of instances */
1490  {
1491          ulint           i;
1492          const ulint     size    = total_size / n_instances;
       ...
1501          for (i = 0; i < n_instances; i++) {
1502                  buf_pool_t*     ptr     = &buf_
1503
1504                  if (buf_pool_init_instance(ptr, size, i) != DB_SUCCESS) {
1505
1506                          /* Free all the instances created so far. */
1507                          buf_pool_free(i);
1508
1509                          return(DB_ERROR);
1510                  }
1511          }
```

Buffer pool init per instance

Buffer Pool Init

- buf/buf0buf.cc: buf_pool_init_instance()

```
1302  buf_pool_init_instance(
1303  /*===================*/
1304          buf_pool_t*     buf_pool,       /*!< in: buffer pool instance */
1305          ulint           buf_pool_size,  /*!< in: size in bytes */
1306          ulint           instance_no)    /*!< in: id of the instance */
1307  {
1308          ulint           i;
1309          buf_chunk_t*    chunk;
1310
1311          /* 1. Initialize general fields
1312          ------------------------------- */
1313          mutex_create(buf_pool_mutex_key,
1314                       &buf_pool->mutex, SYNC_BUF_POOL);
1315          mutex_create(buf_pool_zip_mutex_key,
1316                       &buf_pool->zip_
1317
1318          buf_pool_mutex_enter(buf_pool);
```

Create mutex for buffer pool

Buffer Pool Init

- buf/buf0buf.cc: buf_pool_init_instance()

```
1320            if (buf_pool_size > 0) {
1321                    buf_pool->n_chunks = 1;
1322
1323                    buf_pool->chunks = chunk =
1324                            (buf_chunk_t*) mem_zalloc(sizeof *chunk);
1325
1326                    UT_LIST_INIT(buf_pool->free);
1327
1328                    if (!buf_chunk_init(buf_pool, chunk, buf_pool_size)) {
1329                            mem_free(chunk);
1330                            mem_free(buf_pool);
1331
1332                            buf_pool_mutex_exit(buf_pool);
1333
1334                            return(DB_ERROR);
1335                    }
```

Initialize buffer chunk

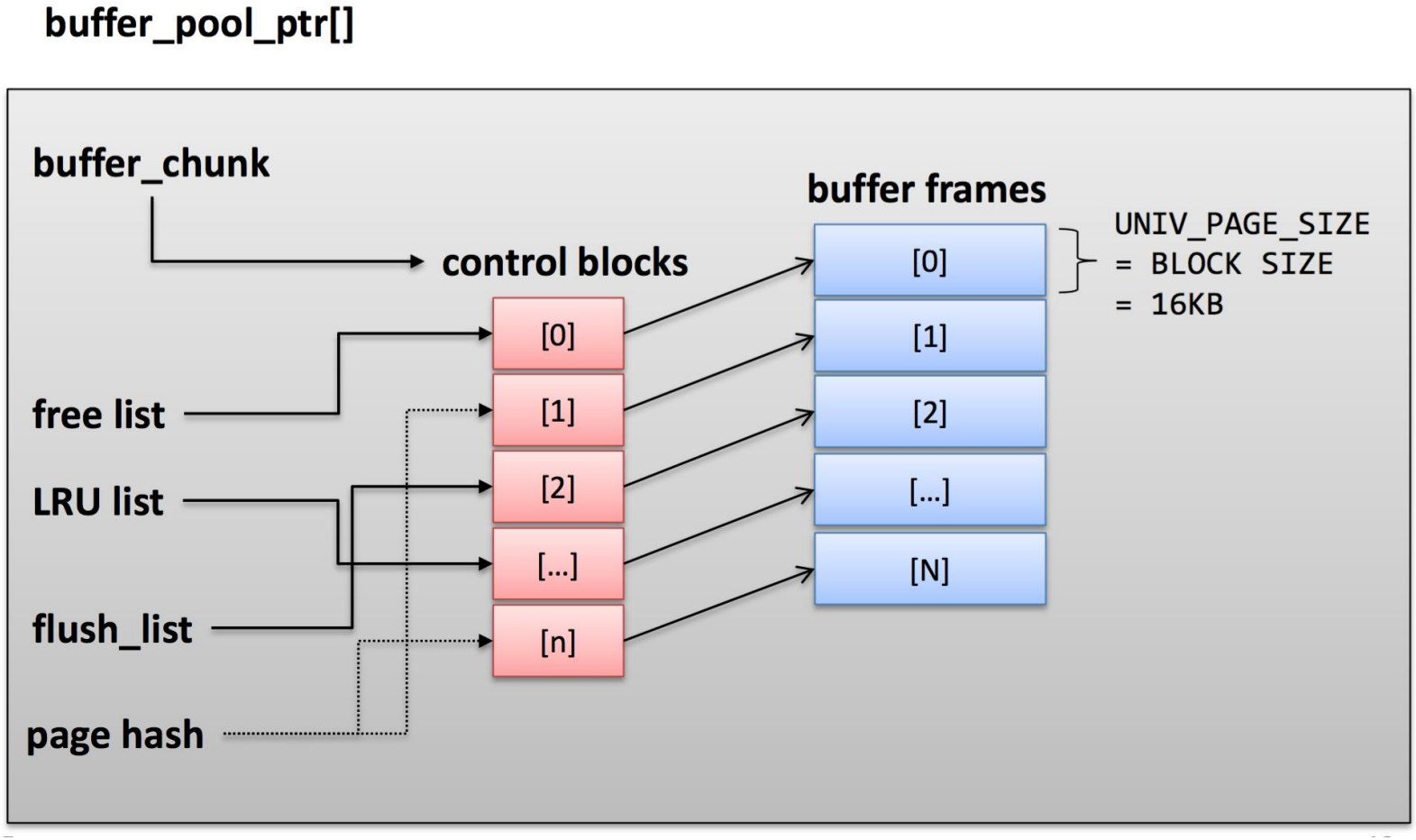
Buffer Pool Init

- buf/buf0buf.cc: buf_pool_init_instance()

```
1349                buf_pool->page_hash = ha_create(2 * buf_pool->curr_size,
1350                                                srv_n_page_hash_locks,
1351                                                MEM_HEAP_FOR_PAGE_HASH,
1352                                                SYNC_BUF_PAGE_HASH);
1353
1354                buf_pool->zip_hash = hash_create(2
1355
1356                buf_pool->last_printout_time = ut_time();
1357        }
1358        /* 2. Initialize flushing fields
1359        -------------------------------- */
1360
1361        mutex_create(flush_list_mutex_key, &buf_pool->flush_list_mutex,
1362                     SYNC_BUF_FLUSH_LIST);
```

Create page hash table

Buffer Chunk



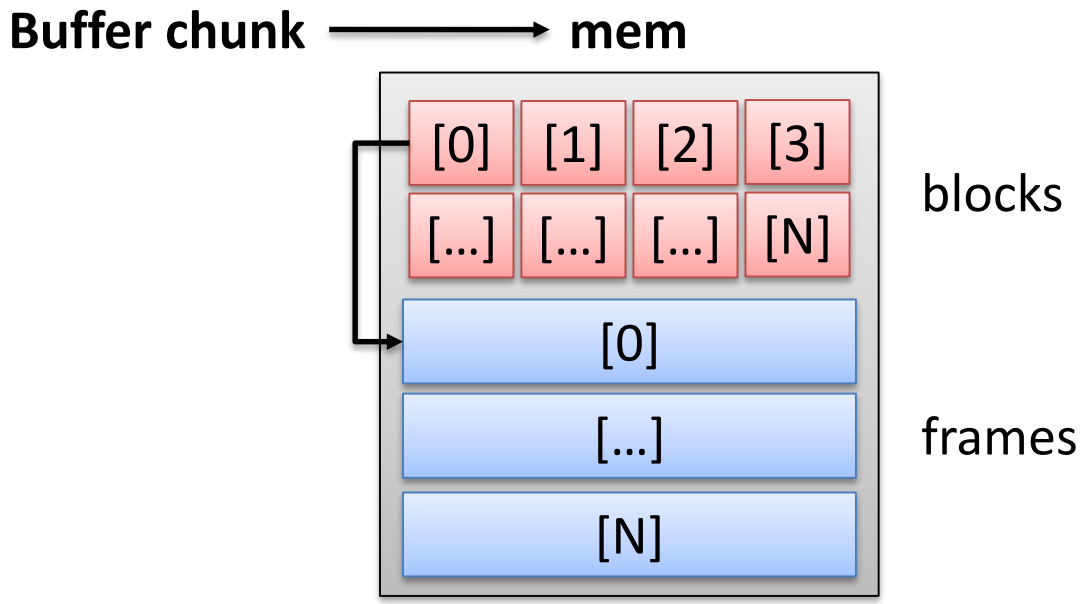
- Total buffer pool size

  = x * innodb_buffer_pool_instances * innodb_buffer_pool_chunk_size

Buffer Chunk Struct

- include/buf0buf.ic: buf_chunk_t

```
39  /** A chunk of buffers. The buffer pool is allocated in chunks. */
40  struct buf_chunk_t{
41          ulint           mem_size;       /*!< allocated size of the chunk */
42          ulint           size;           /*!< size of frames[] and blocks[] */
43          void*           mem;            /*!< pointer to the memory area which
44                                          was allocated for the frames */
45          buf_block_t*    blocks;         /*!< array of buffer control blocks */
46  };
```



Buffer Pool Init

- buf/buf0buf.cc: buf_chunk_init()

```
1090  buf_chunk_init(
1091  /*============*/
1092          buf_pool_t*     buf_pool,       /*!< in: buffer pool instance */
1093          buf_chunk_t*    chunk,          /*!< out: chunk of buffers */
1094          ulint           mem_size)       /*!< in: requested size in bytes */
1095  {
1096          buf_block_t*    block;
1097          byte*           frame;
1098          ulint           i;
1099
1100          /* Round down to a multiple of page size,
1101          although it already should be. */
1102          mem_size = ut_2pow_round(mem_size, UNIV_PAGE_SIZE);
1103          /* Reserve space for the block descriptors. */
1104          mem_size += ut_2pow_round((mem_size / UNIV_PAGE_SIZE) * (sizeof *block)
1105                                    + (UNIV_PAGE_SIZE - 1), UNIV_PAGE_SIZE);
1106
1107          chunk->mem_size = mem_size;
1108          chunk->mem = os_mem_alloc_large(&chunk->mem_size);
```

Allocate chunk mem (blocks + frames)

Buffer Pool Init

- buf/buf0buf.cc: buf_chunk_init()

```
1115            /* Allocate the block descrip
1116            the start of the memory block. */
1117            chunk->blocks = (buf_block_t*) chunk->mem;
1118
1119            /* Align a pointer to the first frame.  Note that when
1120            os_large_page_size is smaller than UNIV_PAGE_SIZE,
1121            we may allocate one fewer block than requested.  When
1122            it is bigger, we may allocate more blocks than requested. */
1123
1124            frame = (byte*) ut_align(chunk->mem, UNIV_PAGE_SIZE);
1125            chunk->size = chunk->mem_size / UNI
1126                    - (frame != chunk->mem);
```

Allocate control blocks

Allocate frames
(Page size is aligned)

Buffer Pool Init

- buf/buf0buf.cc: buf_chunk_init()

```
1140        /* Init block structs and assign frames for them. Then we
1141        assign the frames to the first blocks (we already mapped the
1142        memory above). */
1143
1144        block = chunk->blocks;
1145
1146        for (i = chunk->size; i--; ) {
1147
1148                buf_block_init(buf_pool, block, frame);
1149                UNIV_MEM_INVALID(block->frame, UNIV_PAGE_SIZE);
1150
1151                /* Add the block to the free list */
1152                UT_LIST_ADD_LAST(list, buf_pool->free, (&block->page));
1153
1154                ut_d(block->page.in_free_list =
1155                ut_ad(buf_pool_from_block(block) == buf_pool);
1156
1157                block++;
1158                frame += UNIV_PAGE_SIZE;
1159        }
```

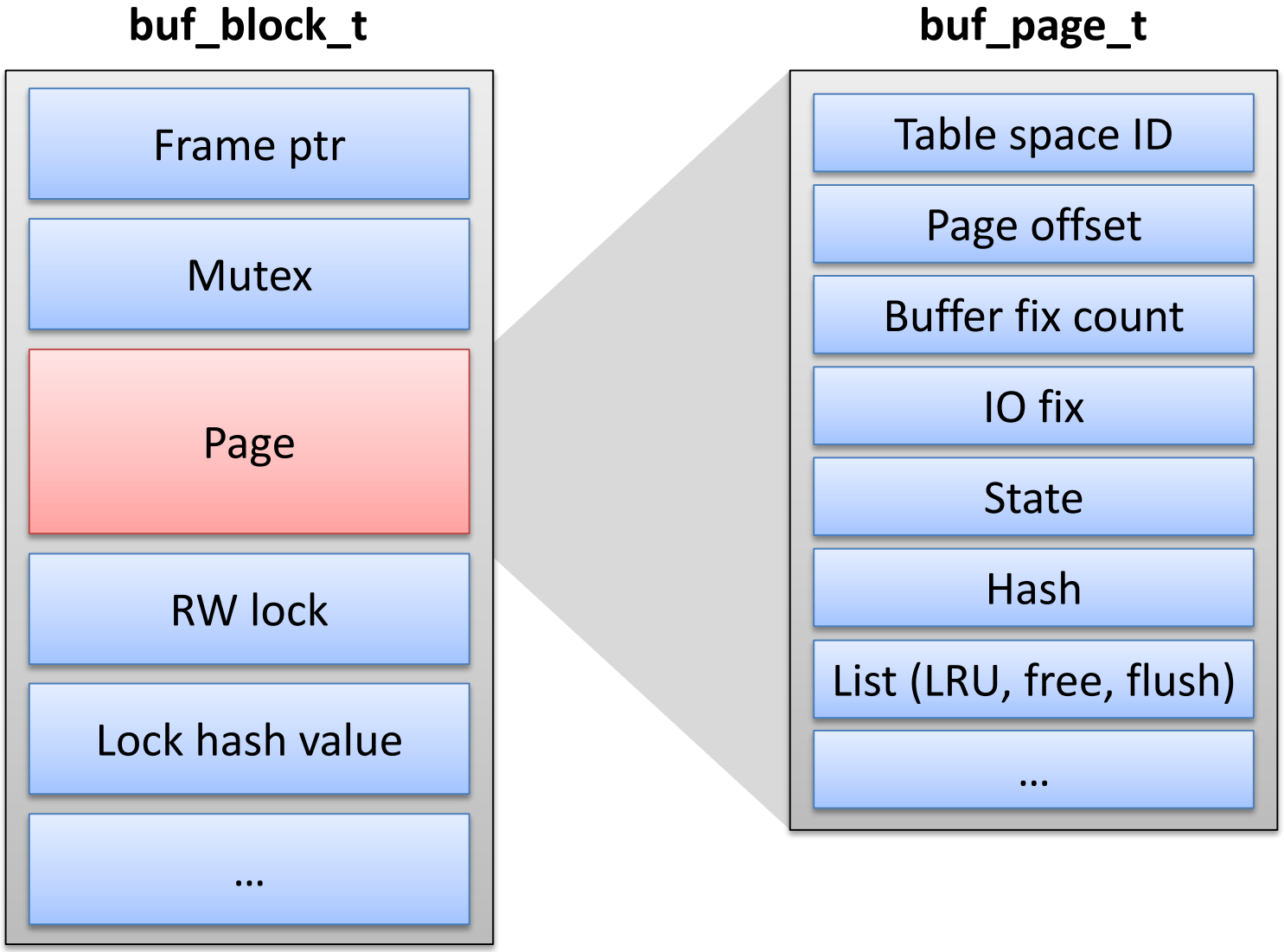
Initialize control block

Add all blocks to free list

Buffer Control Block (BCB)

- The control block contains
  - Read-write lock
  - Buffer fix count
    - Which is incremented when a thread wants a file page to be fixed in a buffer frame
    - The buffer fix operation does not lock the contents of the frame
  - Page frame
  - File pages
    - Put to a hash table according to the file address of the page

Buffer Control Block (BCB)



BCB Struct

- include/buf0buf.h: buf_block_t

```
1624  struct buf_block_t{
1625
1626          /** @name General fields */
1627          /* @{ */
1628
1629          buf_page_t      page;           /*!< page information; this must
1630                                          be the first field, so that
1631                                          buf_pool->page_hash can point
1632                                          to buf_page_t or buf_block_t */
1633          byte*           frame;          /*!< pointer to buffer frame which
1634                                          is of size UNIV_PAGE_SIZE, and
1635                                          aligned to an address divisible by
1636                                          UNIV_PAGE_SIZE */
```

BCB Struct

- include/buf0buf.h: buf_block_t

```
1648        ib_mutex_t      mutex;          /*!< mutex protecting this block:
1649                                        state (also protected by the buffer
1650                                        pool mutex), io_fix, buf_fix_count,
1651                                        and accessed; we introduce this new
1652                                        mutex in InnoDB-5.1 to relieve
1653                                        contention on the buffer pool mutex */
1654        rw_lock_t       lock;           /*!< read-write lock of the buffer
1655                                        frame */
1656        unsigned        lock_hash_val:32;/*!< hashed value of the page address
1657                                        in the record lock hash table;
1658                                        protected by buf_block_t::lock
1659                                        (or buf_block_t::mutex, buf_pool->mutex
1660                                        in buf_page_get_gen(),
1661                                        buf_page_init_for_read()
1662                                        and buf_page_create()) */
```

BCB Struct

- include/buf0buf.h: buf_page_t

```
1447  struct buf_page_t{
1448          /** @name General fields
1449          None of these bit-fields must be modified without holding
1450          buf_page_get_mutex() [buf_block_t::mutex or
1451          buf_pool->zip_mutex], since they can be stored in the same
1452          machine word.  Some of these fields are additionally protected
1453          by buf_pool->mutex. */
1454          /* @{ */
1455
1456          ib_uint32_t     space;          /*!< tablespace id; also protected
1457                                          by buf_pool->mutex. */
1458          ib_uint32_t     offset;         /*!< page number; also protected
1459                                          by buf_pool->mutex. */
1460          /** count of how manyfold this block is currently bufferfixed */
1461  #ifdef PAGE_ATOMIC_REF_COUNT
1462          ib_uint32_t     buf_fix_count;
```

Page identification

BCB Struct

- include/buf0buf.h: buf_page_t

```
1466            byte            io_fix;
1467
1468            byte            state;
 ...
1488            unsigned        flush_type:2;   /*!< if this block is currently being
1489                                            flushed to disk, this tells the
1490                                            flush_type.
1491                                            @see buf_flush_t */
1492            unsigned        buf_pool_index:6;/*!< index number of the buffer pool
1493                                            that this block belongs to */
 ...
1506          buf_page_t*       hash;           /*!< node used in chaining to
1507                                            buf_pool->page_hash or
1508                                            buf_pool->zip_hash */
```

BCB Struct

- include/buf0buf.h: buf_page_t

```
1562        lsn_t           newest_modification;
1563                                /*!< log sequence number of
1564                                the youngest modification to
1565                                this block, zero if not
1566                                modified. Protected by block
1567                                mutex */
1568        lsn_t           oldest_modification;
1569                                /*!< log sequence number of
1570                                the START of the log entry
1571                                written of the oldest
1572                                modification to this block
1573                                which has not yet been flushed
1574                                on disk; zero if all
1575                                modifications are on disk.
1576                                Writes to this field must be
1577                                covered by both block->mutex
1578                                and buf_pool->flush_list_mutex. Hence
1579                                reads can happen while holding
1580                                any one of the two mutexes */
```

BCB Init

- buf/buf0buf.cc: buf_block_init()

```
1020  buf_block_init(
1021  /*===========*/
1022          buf_pool_t*     buf_pool,       /*!< in: buffer pool instance */
1023          buf_block_t*    block,          /*!< in: pointer to control block */
1024          byte*           frame)          /*!< in: pointer to buffer frame */
1025  {
1026          UNIV_MEM_DESC(frame, UNIV_PAGE_SIZE);
1027
1028          block->frame = frame;
1029
1030          block->page.buf_pool_index = buf_pool_index(buf_pool);
1031          block->page.state = BUF_BLOCK_NOT_USED;
1032          block->page.buf_fix_count = 0;
1033          block->page.io_fix = BUF_IO_NONE;
 ...
1064          mutex_create(PFS_NOT_INSTRUMENTED, &block->mutex, SYNC_BUF_BLOCK);
1065          rw_lock_create(PFS_NOT_INSTRUMENTED, &block->lock, SYNC_LEVEL_VARYING);
```

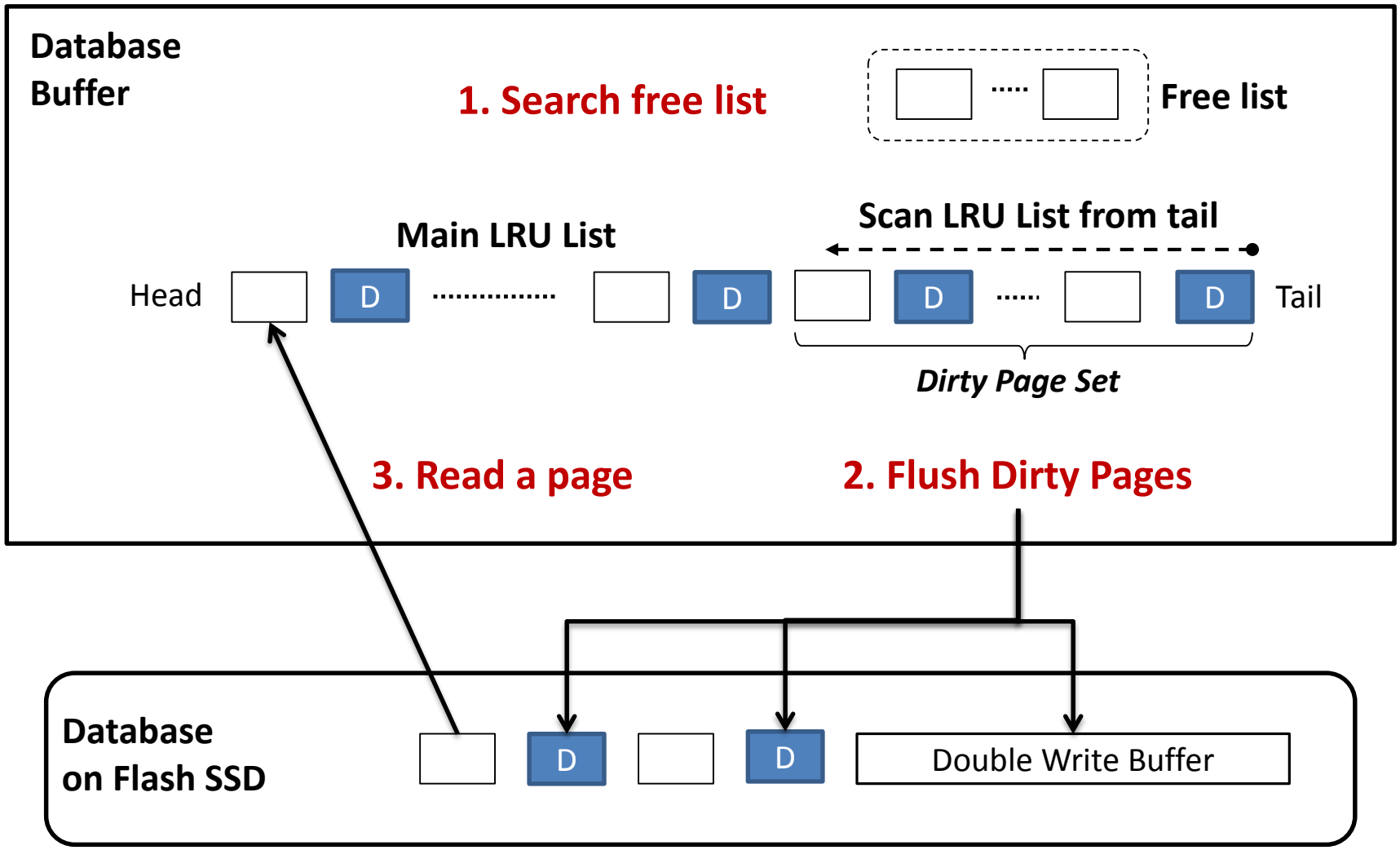
Set data frame

Create block mutex & rw lock

BUFFER READ

PART 1 : READ A PAGE

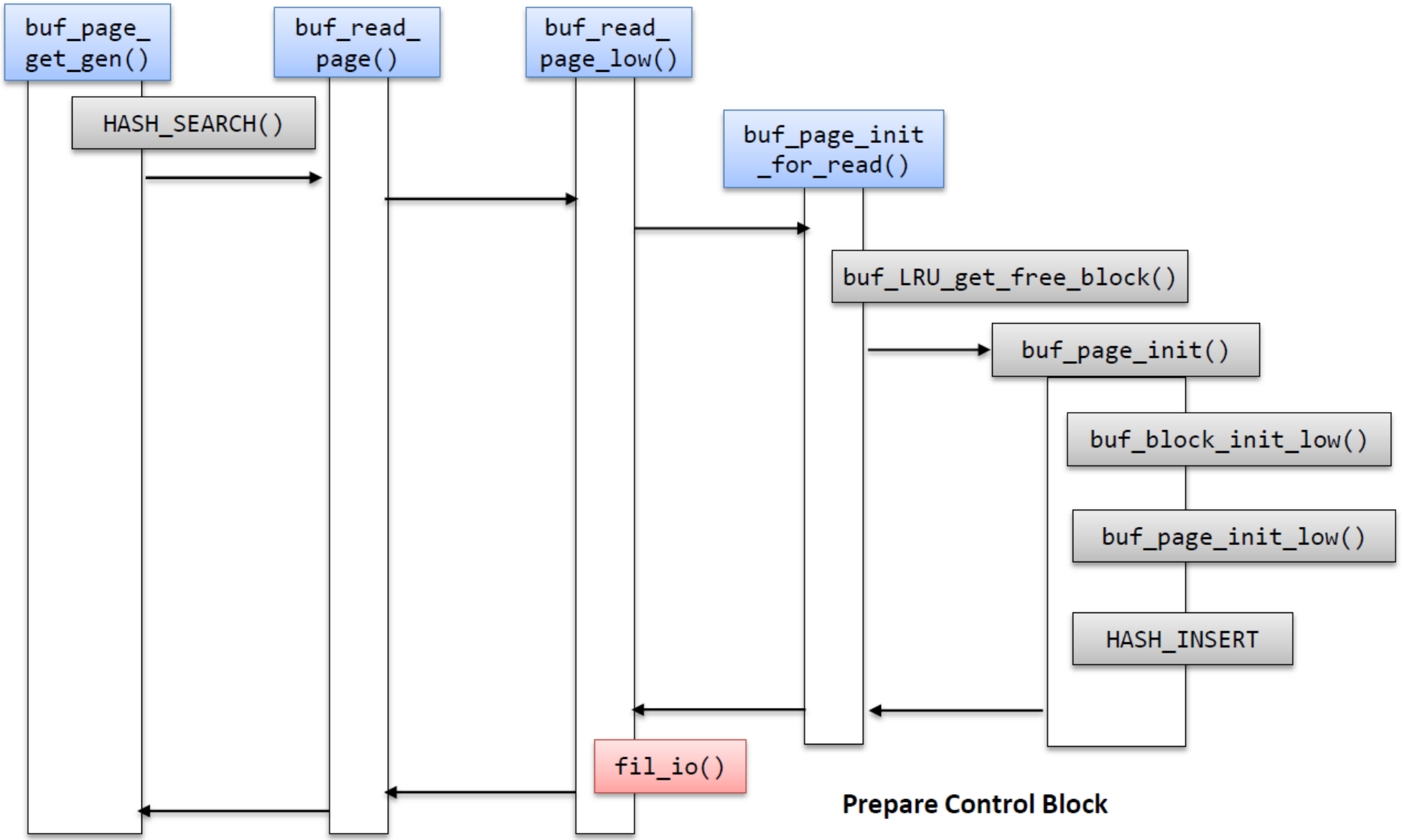
MySQL Buffer Manager: Read



Buffer Read

- Read a page (buf0rea.cc)
  - Find a certain page in the buffer pool using **hash table**
  - If it is not in the buffer pool, then read a block from the storage
    - Allocate a free block for read (include buffer block)
    - Two cases
      - Buffer pool has free blocks
      - Buffer pool doesn't have a free block
    - Read a page

Buffer Read


buf_page_
get_gen()
buf_read_
page()
buf_read_
page_low()
HASH_SEARCH()
buf_page_init
_for_read()
buf_LRU_get_free_block()
buf_page_init()
buf_block_init_low()
buf_page_init_low()
HASH_INSERT
fil_io()
Prepare Control Block

Buffer Read

- buf/buf0buf.cc: buf_page_get_gen()

```
2596  buf_page_get_gen(
2597  /*=============*/
2598          ulint           space,  /*!< in: space id */
2599          ulint           zip_size,/*!< in: compressed page size in bytes
2600                                  or 0 for uncompressed pages */
2601          ulint           offset, /*!< in: page number */
2602          ulint           rw_latch,/*!< in: RW_S_LATCH, RW_X_LATCH, RW_NO_LATCH */
2603          buf_block_t*    guess,  /*!< in: guessed block or NULL */
2604          ulint           mode,   /*!< in: BUF_GET, BUF_GET_IF_IN_POOL,
2605                                  BUF_PEEK_IF_IN_POOL, BUF_GET_NO_LATCH, or
2606                                  BUF_GET_IF_IN_POOL_OR_WATCH */
2607          const char*     file,   /*!< in: file name */
2608          ulint           line,   /*!< in: line where called */
2609          mtr_t*          mtr)    /*!< in: mini-transaction */
2610  {
2611          buf_block_t*    bloc
2612          ulint           fold
2613          unsigned        acce
2614          ulint           fix_
2615          rw_lock_t*      hash
2616          ulint           retr
2617          buf_block_t*    fix_block;
2618          ib_mutex_t*     fix_mutex = NULL;
2619          buf_pool_t*     buf_pool = buf_pool_get(space, offset);
```

Get the buffer pool ptr using space & offset

**** 2 important things ****

1) ID of a page is (*space, offset*) of the page
2) *Buffer pool – page mapping* is mapped

Buffer Read

- include/buf0buf.ic: buf_pool_get()

```
1097  buf_pool_get(
1098  /*==========*/
1099          ulint   space,  /*!< in: space id */
1100          ulint   offset) /*!< in: offset of the page within space */
1101  {
1102          ulint   fold;
1103          ulint   index;
1104          ulint   ignored_offset;
1105
1106          ignored_offset = offset >> 6; /* 2log of BUF_READ_AHEAD_AREA (64)*/
1107          fold = buf_page_address_fold(space, ignored_offset);
1108          index = fold % srv_buf_pool_instances;
1109          return(&buf_pool_ptr[index]);
1110  }
```

Make a fold number

Buffer Read

- Why fold?
  - They want to put pages together in the same buffer pool if it is the **same extents** for read ahead

Buffer Read

- buf/buf0buf.cc: buf_page_get_gen()

```
2648            buf_pool->stat.n_page_gets++;
2649            fold = buf_page_address_fold(space, offset);
2650            hash_lock = buf_page_hash_lock_get(buf_pool, fold);
2651    loop:
2652            block = guess;
2653
2654            rw_lock_s_lock(hash_lock);
```

Get page hash lock before searching in the hash table

Set shared lock on hash table

Buffer Read

- buf/buf0buf.cc: buf_page_get_gen()

```
2675        if (block == NULL) {
2676                block = (buf_block_t*) buf_page_hash_get_low(
2677                        buf_pool, space, offset, fold);
2678        }
...
2685        if (block == NULL) {
2686                /* Page not in buf_pool: needs to be read from file */
...
2718                if (buf_read_page(space, zip_size, offset)) {
2719                        buf_read_ahead_random(space, zip_size, offset
2720
2721
2722                        retries = 0;
2723                } else if (retries < BUF_PAGE_READ_MAX_RETRIES) {
2724                        ++retries;
2725                        DBUG_EXECUTE_IF(
2726                                "innodb_page_corruption_retries",
2727                                retries = BUF_PAGE_READ_MAX_RETRIES;
2728                        );
```

Find a page in the hash table

Page doesn't exist in buffer pool

success

Read the page from the storage

retry

Buffer Read

- buf/buf0buf.cc: buf_page_get_gen()

```
2729            } else {
2730                    fprintf(stderr, "InnoDB: Error: Unable"
2731                            " to read tablespace %lu page no"
2732                            " %lu into the buffer pool after"
2733                            " %lu attempts\n"
2734                            "InnoDB: The most probable cause"
2735                            " of this error may be that the"
2736                            " table has been corrupted.\n"
2737                            "InnoDB: You can try to fix this"
2738                            " problem by using"
2739                            " innodb_force_recovery.\n"
2740                            "InnoDB: Please see reference manual"
2741                            " for more details.\n"
2742                            "InnoDB: Aborting...\n",
2743                            space, offset,
2744                            BUF_PAGE_READ_MAX_RETRIES);
2745
2746                    ut_error;
2747            }
```

Fail

Buffer Read

- buf/buf0buf.cc: buf_page_get_gen()

```
2749  #if defined UNIV_DEBUG || defined UNIV_BUF_DEBUG
2750                  ut_a(++buf_dbg_counter % 5771 || buf_validate());
2751  #endif /* UNIV_DEBUG || UNIV_B
2752                  goto loop;
2753          } else {
2754                  fix_block = block;
2755          }
2756
2757          buf_block_fix(fix_block);
2758
2759          /* Now safe to release page_hash mutex */
2760          rw_lock_s_unlock(hash_lock);
```

If it failed to read target page, go to the first part of the loop

Buffer Read

- buf/buf0rea.cc: buf_read_page()

```
418  buf_read_page(
419  /*==========*/
420          ulint   space,  /*!< in: space id */
421          ulint   zip_size,/*!< in: compressed page size in bytes, or 0 */
422          ulint   offset) /*!< in: page number */
423  {
424          ib_int64_t      tablespace_version;
425          ulint           count;
426          dberr_t         err;
427
428          tablespace_version = fil_space_get_version(space);
429
430          /* We do the i/o in the synchronous aio mode to save thread
431          switches: hence TRUE */
432
433          count = buf_read_page_low(&err, true, BUF_READ_ANY_PAGE, space,
434                                    zip_size, FALSE,
435                                    tablespace_version, offset);
```

Buffer Read

- buf/buf0rea.cc: buf_read_page_low()

```
105  buf_read_page_low(
106  /*==============*/
107          dberr_t*        err,    /*!< out: DB_SUCCESS or DB_TABLESPACE_DELETED if we are
108                                  trying to read from a non-existent tablespace, or a
109                                  tablespace which is just now being dropped */
110          bool    sync,   /*!< in: true if synchronous aio is desired */
111          ulint   mode,   /*!< in: BUF_READ_IBUF_PAGES_ONLY, ...,
112                                  ORed to OS_AIO_SIMULATED_WAKE_LATER (see below
113                                  at read-ahead functions) */
114          ulint   space,  /*!< in: space id */
115          ulint   zip_size,/*!< in: compressed page size, or 0 */
116          ibool   unzip,  /*!< in: TRUE=request uncompressed page */
117          ib_int64_t tablespace_version, /*!< in: if the space memory object has
118                                  this timestamp different from what we are giving here,
119                                  treat the tablespace as dropped; this is a timestamp we
120                                  use to stop dangling page reads from a tablespace
121                                  which we have DISCARDed + IMPORTed back */
122          ulint   offset) /*!< in: page number */
123  {
124          buf_page_t*     bpage;
125          ulint           wake_later;
```

Buffer Read

- buf/buf0rea.cc: buf_read_page_low()

```
158         /* The following call will also check if the tablespace does not exist
159         or is being dropped; if we succeed in initing the page in the buffer
160         pool for read, then DISCARD cannot proceed until the read has
161         completed */
162         bpage = buf_page_init_for_read(err, mode, space, zip_size, unzip,
163                                        tablespace_version, offset);
164         if (bpage == NULL) {
165
166                 return(0);
167         }
```

Allocate buffer block for read

Buffer Read

- buf/buf0buf.cc: buf_page_init_for_read()

```
3586  buf_page_init_for_read(
3587  /*===================*/
3588          dberr_t*        err,    /*!< out: DB_SUCCESS or DB_TABLESPACE_DELETED */
3589          ulint           mode,   /*!< in: BUF_READ_IBUF_PAGES_ONLY, ... */
3590          ulint           space,  /*!< in: space id */
3591          ulint           zip_size,/*!< in: compressed page size, or 0 */
3592          ibool           unzip,  /*!< in: TRUE=request uncompressed page */
3593          ib_int64_t      tablespace_version,
3594                                  /*!< in: prevents reading from a wrong
3595                                  version of the tablespace in case we have done
3596                                  DISCARD + IMPORT */
3597          ulint           offset) /*!< in: page number */
3598  {
3599          buf_block_t*    block;
3600          buf_page_t*     bpage   = NULL;
3601          buf_page_t*     watch_page;
3602          rw_lock_t*      hash_lock;
3603          mtr_t           mtr;
3604          ulint           fold;
3605          ibool           lru     = FALSE;
3606          void*           data;
3607          buf_pool_t*     buf_pool = buf_pool_get(space, offset);
```

Buffer Read

- buf/buf0buf.cc: buf_page_init_for_read()

```
3631            if (zip_size && !unzip && !recv_recovery_is_on()) {
3632                    block = NULL;
3633            } else {
3634
3635            block = buf_LRU_get_free_block(buf_pool);
3636
3637            ut_ad(block);
3638                    ut_ad(buf_pool_from_block(block) == buf_pool);
3639            }
3640
3641            fold = buf_page_address_fold(space, offset);
3642            hash_lock = buf_page_hash_lock_get(buf_pool, fold);
3643
3644            buf_pool_mutex_enter(buf_pool);
3645            rw_lock_x_lock(hash_lock);
```

Get free block: see this later

Buffer Read

- buf/buf0buf.cc: buf_page_init_for_read()

```
3672            if (block) {
3673                    bpage = &block->page;
3674
3675                    mutex_enter(&block->mutex);
3676
3677                    ut_ad(buf_pool_from_bpage(bpage) == buf_pool);
3678
3679                    buf_page_init(buf_pool, space, offset, fold, zip_size, block);
3680
3681    #ifdef PAGE_ATOMIC_REF_COUNT
3682                    /* Note: We set the io stat
3683                    the block->lock. This is because other threads cannot
3684                    access this block unless it is in the hash table. */
3685
3686                    buf_page_set_io_fix(bpage, BUF_IO_READ);
3687    #endif /* PAGE_ATOMIC_REF_COUNT */
3688
3689                    rw_lock_x_unlock(hash_lock);
3690
3691                    /* The block must be put to the LRU list, to the old blocks */
3692                    buf_LRU_add_block(bpage, TRUE/* to old blocks */);
```

Initialize buffer page for current read

Buffer Read

- buf/buf0buf.cc: buf_page_init()

```
3485  buf_page_init(
3486  /*==========*/
3487          buf_pool_t*     buf_pool,/*!< in/out: buffer pool */
3488          ulint           space,  /*!< in: space id */
3489          ulint           offset, /*!< in: offset of the page within space
3490                                  in units of a page */
3491          ulint           fold,   /*!< in: buf_page_address_fold(space,offset) */
3492          ulint           zip_size,/*!< in: compressed page size, or 0 */
3493          buf_block_t*    block)  /*!< in/out: block to init */
3494  {
3495          buf_page_t*     hash_page;
3496
3497          ut_ad(buf_pool == buf_pool_get(space, offset));
3498          ut_ad(buf_pool_mutex_own(buf_pool));
3499
3500          ut_ad(mutex_own(&(block->mutex)));
3501          ut_a(buf_block_get_state(block) != BUF_BLOCK_FILE_PAGE);
3502
3503  #ifdef UNIV_SYNC_DEBUG
3504          ut_ad(rw_lock_own(buf_page_hash_lock_get(buf_pool, fold),
3505                            RW_LOCK_EX));
3506  #endif /* UNIV_SYNC_DEBUG */
```

Buffer Read

- buf/buf0buf.cc: buf_page_init()

```
3508            /* Set the state of the block */
3509            buf_block_set_file_page(block, space, offset);
3510
3511    #ifdef UNIV_DEBUG_VALGRIND
3512            if (!space) {
3513                    /* Silence valid Valgrind warnings about uninitialized
3514                    data being written to data files.  There are some unused
3515                    bytes on some pages that InnoDB does not initialize. */
3516                    UNIV_MEM_VALID(block->frame, UNIV_PAGE_SIZE);
3517            }
3518    #endif /* UNIV_DEBUG_VALGRIND */
3519
3520            buf_block_init_low(block);
3521
3522            block->lock_hash_val = lock_rec_hash(space, offset);
3523
3524            buf_page_init_low(&block->page);
```

Buffer Read

- buf/buf0buf.cc: buf_page_init()

```
3567            HASH_INSERT(buf_page_t, hash, buf_pool->page_hash, fold, &block->page);
3568
3569            if (zip_size) {
3570                    page_zip_set_size(&block->page.zip, zip_size);
3571            }
3572    }
```

Insert a page into hash table

Buffer Read

- buf/buf0buf.cc: buf_page_init_for_read()

```
3679                    buf_page_init(buf_pool, space, offset, fold, zip_size, block);
3680
3681    #ifdef PAGE_ATOMIC_REF_COUNT
3682                    /* Note: We set the io state without the protection of
3683                    the block->lock. This is because other threads cannot
3684                    access this block unless i
3685
3686                    buf_page_set_io_fix(bpage, BUF_IO_READ);
3687    #endif /* PAGE_ATOMIC_REF_COUNT */
3688
3689                    rw_lock_x_unlock(hash_lock);
3690
3691                    /* The block must be put to the LRU list, to the old blocks */
3692                    buf_LRU_add_block(bpage, TRUE/* to old blocks */);
```

Set io fix to BUF_IO_READ

Add current block to LRU list
: see this later

Buffer Read

- buf/buf0buf.cc: buf_page_init_for_read()

```
3832            buf_pool->n_pend_reads++;
3833    func_exit:
3834            buf_pool_mutex_exit(buf_pool
3835
3836            if (mode == BUF_READ_IBUF_PAGES_ONLY) {
3837
3838                    ibuf_mtr_commit(&mtr);
3839            }
3840
3841
3842    #ifdef UNIV_SYNC_DEBUG
3843            ut_ad(!rw_lock_own(hash_lock, RW_LOCK_EX));
3844            ut_ad(!rw_lock_own(hash_lock, RW_LOCK_SHARED));
3845    #endif /* UNIV_SYNC_DEBUG */
3846
3847            ut_ad(!bpage || buf_page_in_file(bpage));
3848            return(bpage);
3849    }
```

Increase *pending read count*; How many buffer read were requested but not finished

Buffer Read

- We allocate a free buffer and control block
- And the block was inserted into hash table and LRU list of corresponding buffer pool

- Now, we need to read the real content of the target page from the **storage**

Buffer Read

- buf/buf0rea.cc: buf_read_page_low()

```
207             if (zip_size) {
208                     *err = fil_io(OS_FILE_READ | wake_later
209                                   | ignore_nonexistent_pages,
210                                   sync, space, zip_size, offset, 0, zip_size,
211                                   bpage->zip.data, bpage);
212             } else {
213                     ut_a(buf_page_get_state(bpage) =
214
215                     *err = fil_io(OS_FILE_READ | wake_later
216                                   | ignore_nonexistent_pages,
217                                   sync, space, 0, offset, 0, UNIV_PAGE_SIZE,
218                                   ((buf_block_t*) bpage)->frame, bpage);
219             }
```

Read a page from storage
: see this later

Buffer Read

- buf/buf0rea.cc: buf_read_page_low()

```
234             if (sync) {
235                     /* The i/o is already completed when we arrive from
236                     fil_read */
237                     if (!buf_page_io_complete(bpage)) {
238                             return(0);
239                     }
240             }
241
242             return(1);
243     }
```

Complete the read request

Buffer Read

- buf/buf0buf.cc: buf_page_io_complete()

```
4162  buf_page_io_complete(
4163  /*=================*/
4164          buf_page_t*     bpage)  /*!< in: pointer to the block in question */
4165  {
4166          enum buf_io_fix io_type;
4167          buf_pool_t*     buf_pool = buf_pool_from_bpage(bpage);
4168          const ibool     uncompressed = (buf_page_get_state(bpage)
4169                                          == BUF_BLOCK_FILE_PAGE);
4170
4171          ut_a(buf_page_in_file(bpage));
4172
4173          /* We do not need protect io_fix here by mutex to read
4174          it because this is the only function where we can change the value
4175          from BUF_IO_READ or BUF_IO_WRITE to some other value, and our code
4176          ensures that this is the only thread that handles the i/o for this
4177          block. */
4178
4179          io_type = buf_page_get_io_fix(bpage);
```

Get io type (In this case, BUF_IO_READ)

Buffer Read

- buf/buf0buf.cc: buf_page_io_complete()

```
4182            if (io_type == BUF_IO_READ) {
...
4243                    if (buf_page_is_corrupted(true, frame,
4244                                              buf_page_get_zip_size(bpage))) {
4245
4246                            /* Not a real corru
4247                            error injection */
4248                            DBUG_EXECUTE_IF("buf_page_is_corrupt_failure",
4249                                    if (bpage->space > TRX_SYS_SPACE
4250                                        && buf_mark_space_corrupt(bpage)) {
4251                                            ib_logf(IB_LOG_LEVEL_INFO,
4252                                                    "Simulated page corruption");
4253                                            return(true);
4254                                    }
4255                                    goto page_not_corrupt;
4256                                    ;);
4257    corrupt:
4258                            fprintf(stderr,
4259                                    "InnoDB: Database page corruption on disk"
4260                                    " or a failed\n"
```

Page corruption check based on checksum in the page

Buffer Read

- buf/buf0buf.cc: buf_page_io_complete()

```
4329            buf_pool_mutex_enter(buf_pool);
4330            mutex_enter(buf_page_get_mutex(bpage));
4331
4332    #ifdef UNIV_IBUF_COUNT_DEBUG
4333            if (io_type == BUF_IO_WRITE || uncompressed) {
4334                    /* For BUF_IO_READ of compressed-only blocks, the
4335                    buffered operations will be merged by buf_page_get_gen()
4336                    after the block has been uncompressed. */
4337                    ut_a(ibuf_count_get(bpage->space, bpage->offset) == 0);
4338            }
4339    #endif
4340            /* Because this thread which does the unlocking is not the same that
4341            did the locking, we use a pass value != 0 in unlock, which simply
4342            removes the newest lock debug record, without checking the thread
4343            id. */
4344
4345            buf_page_set_io_fix(bpage, BUF_IO_NONE);
```

Set io fix to BUF_IO_NONE

Buffer Read

- buf/buf0buf.cc: buf_page_io_complete()

```
4347            switch (io_type) {
4348            case BUF_IO_READ:
4349                    /* NOTE that the call to ibuf may have moved the ownership of
4350                    the x-latch to this OS thread: do not let this confuse you in
4351                    debugging! */
4352
4353                    ut_ad(buf_pool->n_pend_reads > 0);
4354                    buf_pool->n_pend_reads--;
4355                    buf_pool->stat.n_pages_read++;
4356
4357                    if (uncompressed) {
4358                            rw_lock_x_unlock_gen(&((buf_block_t*) bpage)->lock,
4359                                                 BUF_IO_READ);
4360                    }
4361
4362                    break;
```

Decrease pending read count

BUFFER READ

PART 2 : AFTER GOT BLOCK

Buffer Read

- buf/buf0buf.cc: buf_page_get_gen()

```
2762  got_block:
2763
2764          fix_mutex = buf_page_get_mutex(&fix_block->page);
 ...
3076          /* Check if this is the first access to the page */
3077          access_time = buf_page_is_accessed(&fix_block->page);
3078
3079          /* This is a heuristic and we don't care about ordering issues. */
3080          if (access_time == 0) {
3081                  buf_block_mutex_enter(fix_block);
3082
3083                  buf_page_set_accessed(&fix_block->page);
3084
3085                  buf_block_mutex_exit(fix_block);
3086          }
```

Set access time

Buffer Read

- buf/buf0buf.cc: buf_page_get_gen()

```
3131            if (mode != BUF_PEEK_IF_IN_POOL && !access_time) {
3132                    /* In the case of a first access, t
3133                    read-ahead */
3134
3135                    buf_read_ahead_linear(
3136                            space, zip_size, offset, ibuf_inside(mtr));
3137            }
3138
3139    #ifdef UNIV_IBUF_COUNT_DEBUG
3140            ut_a(ibuf_count_get(buf_block_get_space(fix_block),
3141                                buf_block_get_page_no(fix_block)) == 0);
3142    #endif
3143    #ifdef UNIV_SYNC_DEBUG
3144            ut_ad(!rw_lock_own(hash_lock, RW_LOCK_EX));
3145            ut_ad(!rw_lock_own(hash_lock, RW_LOCK_SHARED));
3146    #endif /* UNIV_SYNC_DEBUG */
3147            return(fix_block);
3148    }
```

Do read ahead process (default=false)

LRU REPLACEMENT

PART 1 : ADD BLOCK

LRU Add Block

- buf/buf0buf.cc: buf_page_init_for_read()

```
3679                    buf_page_init(buf_pool, space, offset, fold, zip_size, block);
3680
3681    #ifdef PAGE_ATOMIC_REF_COUNT
3682                    /* Note: We set the io state without the protection of
3683                    the block->lock. This is because other threads cannot
3684                    access this block unless it is in the hash table. */
3685
3686                    buf_page_set_io_fix(bpage, BUF_IO_READ);
3687    #endif /* PAGE_ATOMIC_REF_COUNT */
3688
3689                    rw_lock_x_unlock(hash_lock);
3690
3691                    /* The block must be put to the LRU list, to the old blocks */
3692                    buf_LRU_add_block(bpage, TRUE/* to old blocks */);
```

Add current block to LRU list

LRU Add Block

- buf/buf0lru.cc: buf_LRU_add_block()

```
1854  buf_LRU_add_block(
1855  /*==============*/
1856          buf_page_t*     bpage,  /*!< in: control block */
1857          ibool           old)    /*!< in: TRUE if should be put to the old
1858                                  blocks in the LRU list, else put to the start;
1859                                  if the LRU list is very short, the block is
1860                                  added to the start, regardless of this
1861                                  parameter */
1862  {
1863          buf_LRU_add_block_low(bpage, old);
1864  }
```

LRU Add Block

- buf/buf0lru.cc: buf_LRU_add_block_low()

```
1782  buf_LRU_add_block_low(
1783  /*======================*/
1784          buf_page_t*     bpage,  /*!< in: control block */
1785          ibool           old)    /*!< in: TRUE if should be put to the old blocks
1786                                  in the LRU list, else put to the start; if the
1787                                  LRU list is very short, the block is added to
1788                                  the start, regardless of this parameter */
1789  {
1790          buf_pool_t*     buf_pool = buf_pool_from_bpage(bpage);
1791
1792          ut_ad(buf_pool_mutex_own(buf_pool));
1793
1794          ut_a(buf_page_in_file(bpage));
1795          ut_ad(!bpage->in_LRU_list);
1796
1797          if (!old || (UT_LIST_GET_LEN(buf_pool->LRU) < BUF_LRU_OLD_MIN_LEN)) {
1798
1799                  UT_LIST_ADD_FIRST(LRU, buf_pool->LRU, bpage);
1800
1801                  bpage->freed_page_clock = buf_pool->freed_page_clock;
1802          } else {
```

If list is too small, then put current block to first of the list

LRU Add Block

- buf/buf0lru.cc: buf_LRU_add_block_low()

```
1802            } else {
1803    #ifdef UNIV_LRU_DEBUG
1804                    /* buf_pool->LRU_old must be the first item in the LRU list
1805                    whose "old" flag is set. */
1806                    ut_a(buf_pool->LRU_old->old);
1807                    ut_a(!UT_LIST_GET_PREV(LRU, buf_pool->LRU_old)
1808                         || !UT_LIST_GET_PREV(LRU, buf_pool->LRU_old)->old);
1809                    ut_a(!UT_LIST_GET_NEXT(LRU, bu
1810                         || UT_LIST_GET_NEXT(LRU,
1811    #endif /* UNIV_LRU_DEBUG */
1812                    UT_LIST_INSERT_AFTER(LRU, buf_pool->LRU, buf_pool->LRU_old,
1813                                         bpage);
1814                    buf_pool->LRU_old_len++;
1815            }
```

Else, insert current block to after LRU_old pointer

LRU REPLACEMENT

PART 2 : GET FREE BLOCK

LRU Get Free Block

- This function is called from a **user thread** when it needs a clean block to read in a page
  - Note that we only ever get a block from the free list
  - Even when we flush a page or find a page in LRU scan we put it to free list to be used

LRU Get Free Block

- iteration 0:
  - get a block from free list, success: done
  - if there is an LRU flush batch in progress:
    - wait for batch to end: retry free list
  - if *buf_pool->try_LRU_scan* is set
    - scan LRU up to ***srv_LRU_scan_depth*** to find a clean block
    - the above will put the block on free list
    - success: retry the free list
  - flush one dirty page from tail of LRU to disk (= **single page flush**)
    - the above will put the block on free list
    - success: retry the free list

LRU Get Free Block

- iteration 1:
  - same as iteration 0 except:
    - scan whole LRU list
- iteration > 1:
  - same as iteration 1 but sleep 100ms

LRU Get Free Block

- buf/buf0lru.cc: buf_LRU_get_free_block()

```
1239  buf_LRU_get_free_block(
1240  /*=====================*/
1241          buf_pool_t*     buf_pool)       /*!< in/out: buffer pool instance */
1242  {
1243          buf_block_t*    block           = NULL;
1244          ibool           freed           = FALSE;
1245          ulint           n_iterations    = 0;
1246          ulint           flush_failures  = 0;
1247          ibool           mon_value_was   = FALSE;
1248          ibool           started_monitor = FALSE;
1249
1250          MONITOR_INC(MONITOR_
1251  loop:
1252          buf_pool_mutex_enter(buf_pool);
1253
1254          buf_LRU_check_size_of_non_data_objects(buf_pool);
1255
1256          /* If there is a block in the free list, take it */
1257          block = buf_LRU_get_free_only(buf_pool);
```

Get buffer pool mutex

Get a free block from the free list

LRU Get Free Block

- buf/buf0lru.cc: buf_LRU_get_free_block()

```
1259        if (block) {
1260
1261                buf_pool_mutex_exit(buf_pool);
1262                ut_ad(buf_pool_from_block(block) == buf_pool);
1263                memset(&block->page.zip, 0, sizeof block->page.zip);
1264
1265                if (started_monitor) {
1266                        srv_print_innodb_monitor =
1267                                static_cast<my_bool>(mon_value_was);
1268                }
1269
1270                return(block);
1271        }
```

Getting a free block succeeded

LRU Get Free Block

- buf/buf0lru.cc: buf_LRU_get_free_block()

```
1273            if (buf_pool->init_flush[BUF_FLUSH_LRU]
1274                && srv_use_doublewrite_buf
1275                && buf_dblwr != NULL) {
1276
1277                    /* If there is an LRU flush happening in the background
1278                    then we wait for it to end instead of trying a single
1279                    page flush. If, however, we are not using doublewrite
1280                    buffer then it is better to do our own single page
1281                    flush instead of waiting for LRU flush to end. */
1282                    buf_pool_mutex_exit(buf_pool);
1283                    buf_flush_wait_batch_end(buf_pool, BUF_FLUSH_LRU);
1284                    goto loop;
1285            }
```

If already background flushed started, wait for it to end

LRU Get Free Block

- buf/buf0lru.cc: buf_LRU_get_free_block()

```
1287            freed = FALSE;
1288            if (buf_pool->try_LRU_scan || n_iterations > 0) {
1289                    /* If no block was in
1290                    end of the LRU list ar
1291                    If we are doing for th
1292                    tail of the LRU list otherwise we scan the whole LRU
1293                    list. */
1294                    freed = buf_LRU_scan_and_free_block(buf_pool,
1295                                                        n_iterations > 0);
1296
1297                    if (!freed && n_iterations == 0) {
1298                            /* Tell other threads that there is no point
1299                            in scanning the LRU list. This flag is set to
1300                            TRUE again when we flush a batch from this
1301                            buffer pool. */
1302                            buf_pool->try_LRU_scan = FALSE;
1303            }
1304            }
```

Find a victim page to replace and make a free block

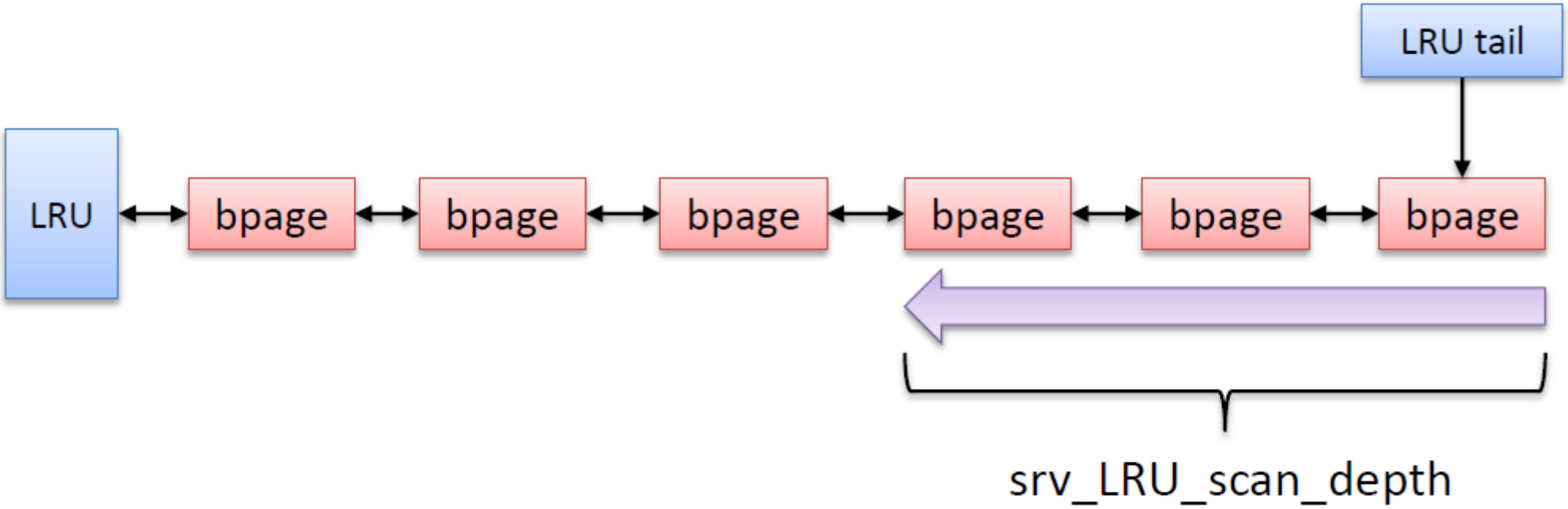
LRU Get Free Block

- buf/buf0lru.cc: buf_LRU_scan_and_free_block()

```
1046  buf_LRU_scan_and_free_block(
1047  /*========================*/
1048          buf_pool_t*     buf_pool,       /*!< in: buffer pool instance */
1049          ibool           scan_all)       /*!< in: scan whole LRU list
1050                                          if TRUE, otherwise scan only
1051                                          'old' blocks. */
1052  {
1053          ut_ad(buf_pool_mutex_own(buf_pool));
1054
1055          return(buf_LRU_free_from_unzip_LRU_list(buf_pool, scan_all)
1056                 || buf_LRU_free_from_common_LRU_list(
1057                         buf_pool, scan_all));
1058  }
```

LRU Get Free Block

- buf/buf0lru.cc: buf_LRU_scan_and_free_block()



LRU Get Free Block

- buf/buf0lru.cc: buf_LRU_free_from_common_LRU_list()

```
1007        for (bpage = UT_LIST_GET_LAST(buf_pool->LRU),
1008             scanned = 1, freed = FALSE;
1009             bpage != NULL && !freed
1010             && (scan_all || scanned < srv_LRU_scan_depth);
1011             ++scanned) {
1012
1013                unsigned        accessed;
1014                buf_page_t*     prev_bpage = UT_LIST_GET_PREV(LRU,
1015                                                              bpage);
1016
1017                ut_ad(buf_page_in_file(bpage));
1018                ut_ad(bpage->in_LRU_list);
1019
1020                accessed = buf_page_is_accessed(bpage);
1021                freed = buf_LRU_free_page(bpage, true);
1022                if (freed && !accessed) {
1023                        /* Keep track of pages that are evicted without
1024                        ever being accessed. This gives us a measure of
1025                        the effectiveness of readahead */
1026                        ++buf_pool->stat.n_ra_pages_evicted;
1027                }
1028
1029                bpage = prev_bpage;
1030        }
```

Try to free it

LRU Get Free Block

- buf/buf0lru.cc: buf_LRU_free_page()

```
1911  buf_LRU_free_page(
1912  /*==============*/
1913          buf_page_t*     bpage,  /*!< in: block to be freed */
1914          bool            zip)    /*!< in: true if should remove also the
1915                                  compressed page of an uncompressed page */
1916  {
1917          buf_page_t*     b = NULL;
1918          buf_pool_t*     buf_pool = buf_pool_from_bpage(bpage);
1919          const ulint     fold = buf_page_address_fold(bpage->space,
1920                                                       bpage->offset);
1921          rw_lock_t*      hash_lock = buf_page_hash_lock_get(buf_pool, fold);
1922
1923          ib_mutex_t*     block_mutex = buf_page_get_mutex(bpage);
1924
1925          ut_ad(buf_pool_mutex_own(buf_pool));
1926          ut_ad(buf_page_in_file(bpage));
1927          ut_ad(bpag
1928
1929          rw_lock_x_lock(hash_lock);
1930          mutex_enter(block_mutex);
```

Get hash lock & block mutex

LRU Get Free Block

- buf/buf0lru.cc: buf_LRU_free_page()

```
1828            if (zip || !bpage->zip.data) {
1829                    /* This would completely free the block. */
1830                    /* Do not completely free dirty blocks. */
1831
1832                    if (bpage->oldest_modification)
1833                            goto func_exit;
1834                    }
1835            } else if (bpage->oldest_modification > 0
1836                       && buf_page_get_state(bpage) != BUF_BLOCK_FILE_PAGE) {
1837
1838                    ut_ad(buf_page_get_state(bpage) == BUF_BLOCK_ZIP_DIRTY);
1839
1840    func_exit:
1841                    rw_lock_x_unlock(hash_lock);
1842                    mutex_exit(block_mutex);
1843                    return(false);
```

If current page is dirty and not flushed to disk yet, exit

LRU Get Free Block

- buf/buf0lru.cc: buf_LRU_free_page()
  - After *func_exit*, we are on **clean** case!

```
2040            buf_pool_mutex_enter(buf_pool);
2041
2042            mutex_enter(block_mutex);
2043            buf_page_unset_sticky(b != NULL ? b : bpage);
2044            mutex_exit(block_mutex);
2045
2046            buf_LRU_block_free_hashed_page((buf_block_t*) bpage);
2047            return(true);
2048    }
```

LRU Get Free Block

- buf/buf0lru.cc: buf_LRU_block_remove_hashed()

```
2128  buf_LRU_block_remove_hashed(
2129  /*========================*/
2130          buf_page_t*     bpage,  /*!< in: block, must contain a file page and
2131                                  be in a state where it can be freed; there
2132                                  may or may not be a hash index to the page */
2133          bool            zip)    /*!< in: true if should remove also the
2134                                  compressed page of an uncompressed page */
2135  {
 ...
2154          buf_LRU_remove_block(bpage);
 ...
2263          HASH_DELETE(buf_page_t, hash, buf_pool->page_hash, fold, bpage);
 ...
2288          case BUF_BLOCK_FILE_PAGE:
2289                  memset(((buf_block_t*) bpage)->frame
2290                         + FIL_PAGE_OFFSET, 0xff, 4);
2291                  memset(((buf_block_t*) bpage)->frame
2292                         + FIL_PAGE_ARCH_LOG_NO_OR_SPACE_ID, 0xff, 4);
2293                  UNIV_MEM_INVALID(((buf_block_t*) bpage)->frame,
2294                                   UNIV_PAGE_SIZE);
2295                  buf_page_set_state(bpage, BUF_BLOCK_REMOVE_HASH);
```

LRU Get Free Block

- buf/buf0lru.cc: buf_LRU_get_free_block()

```
1306            buf_pool_mutex_exit(buf_pool);
1307
1308            if (freed) {
1309                    goto loop;
1310
1311            }
 ...
1352            if (n_iterations > 1) {
1353                    os_thread_sleep(100000);
1354            }
```

If we have free block(s), go to loop

LRU Get Free Block

- buf/buf0lru.cc: buf_LRU_get_free_block()

If we failed to make a free block, do a single page flush

```
1365            if (!buf_flush_single_page_from_LRU(buf_pool)) {
1366                    MONITOR_INC(MONITOR_LRU_SINGLE_FLUSH_FAILURE_COUNT);
1367                    ++flush_failures;
1368            }
1369
1370            srv_stats.buf_pool_wait_free.add(n_iterations, 1);
1371
1372            n_iterations++;
1373
1374            goto loop;
1375    }
```

FLUSH A PAGE

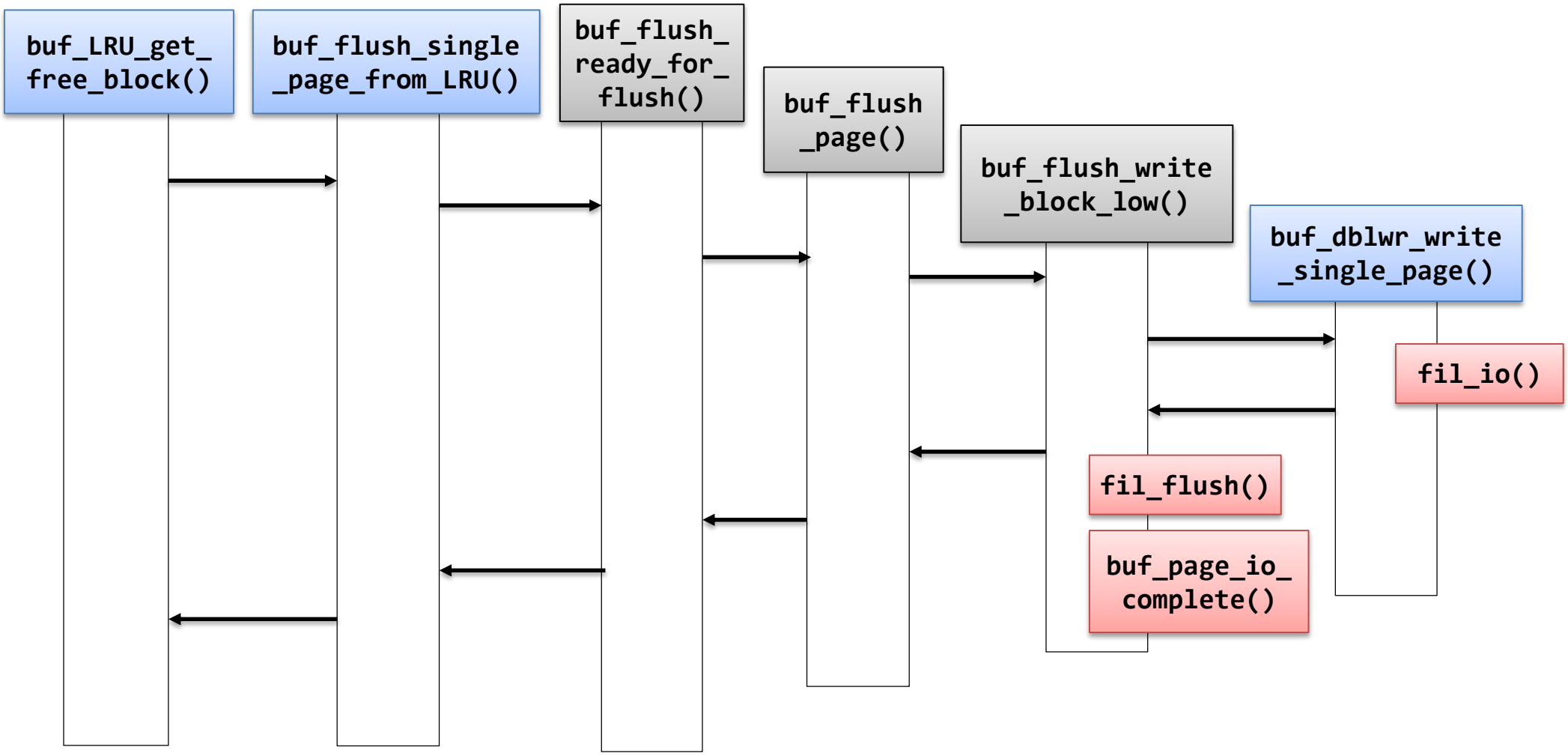
PART 1 : SINGLE PAGE FLUSH

Flush a Page

- Flushing a page by
  - A **background flusher** → batch flush (LRU & flush list)
  - A **single page flush** → in *LRU_get_free_block()*

- Background flusher
  - Regularly check system status (per 1000ms)
  - Flush all buffer pool instances in a batch manner

Single Page Flush



Single Page Flush

- buf/buf0flu.cc: buf_flush_single_page_from_LRU()

```
2018  buf_flush_single_page_from_LRU(
2019  /*===========================*/
2020          buf_pool_t*     buf_pool)       /*!< in/out: buffer pool instance */
2021  {
2022          ulint           scanned;
2023          buf_page_t*     bpage;
2024
2025          buf_pool_mutex_enter(buf_pool);
2026
2027          for (bpage = UT_LIST_GET_LAST(buf_pool->LRU), scanned = 1;
2028               bpage != NULL;
2029               bpage = UT_LIST_GET_PREV(LRU, bpage), ++scanned) {
2030
2031                  ib_mutex_t*     block_mutex = buf_page_get_mutex(bpage);
2032
2033                  mutex_enter(block_mutex);
```

Full scan

Single Page Flush

- buf/buf0flu.cc: buf_flush_single_page_from_LRU()

```
2035            if (buf_flush_ready_for_flush(bpage, BUF_FLUSH_SINGLE_PAGE)) {
2036
2037                    /* The following call
2038                    and block mutex. */
2039
2040                    ibool   flushed = buf_flush_page(
2041                            buf_pool, bpage, BUF_FLUSH_SINGLE_PAGE, true);
2042
2043                    if (flushed) {
2044                            /* buf_flush_page() will release
2045                            block mutex */
2046                            break;
2047                    }
2048            }
2049
2050            mutex_exit(block_mutex);
2051    }
```

Check whether we can flush current block and ready for flush

Try to flush it; write to disk

Single Page Flush

- buf/buf0flu.cc: buf_flush_ready_for_flush()

```
532  buf_flush_ready_for_flush(
533  /*======================*/
534          buf_page_t*     bpage,  /*!< in: buffer control block, must be
535                                  buf_page_in_file(bpage) */
536          buf_flush_t     flush_type)/*!< in: type of flush */
537  {
...
547          if (bpage->oldest_modification == 0
548              || buf_page_get_io_fix(bpage) != BUF_IO_NONE) {
549                  return(false);
550          }
```

If the page is already flushed or doing IO, return false

Single Page Flush

- buf/buf0flu.cc: buf_flush_page()
  - Writes a flushable page from the buffer pool to a file

```
1042  buf_flush_page(
1043  /*===========*/
1044          buf_pool_t*     buf_pool,       /*!< in: buffer pool instance */
1045          buf_page_t*     bpage,          /*!< in: buffer control block */
1046          buf_flush_t     flush_type,     /*!< in: type of flush */
1047          bool            sync)           /*!< in: true if sync IO request */
1048  {
 ...
1077          rw_lock = &reinterpret_cast<buf_block_t*>(bpage)->lock;
1078
1079          if (flush_type != BUF_FLUSH_LIST) {
1080              flush = rw_lock_s_lock_gen_nowait(
1081                      rw_lock, BUF_IO_WRITE);
1082          } else {
1083              /* Will S lock later */
1084              flush = TRUE;
1085          }
```

Get lock

Single Page Flush

- buf/buf0flu.cc: buf_flush_page()

```
1088        if (flush) {
1089            /* We are committed to flushing by the time we get here */
1090
1091            buf_page_set_io_fix(bpage, BUF_IO_WRITE);
1092            buf_page_set_flush_type(bpage, flush_type);
1093
1094            if (buf_pool->n_flush[flush_type] == 0) {
1095                os_event_reset(buf_pool->no_flush[flush_type]);
1096            }
1097
1098            ++buf_pool->n_flush[flush_type];
1099
1100            mutex_exit(block_mutex);
1101            buf_pool_mutex_exit(buf_pool);
 ...
1119            buf_flush_write_block_low(bpage, flush_type, sync);
1120        }
1121
1122            return(flush);
1123    }
```

Set fix and flush type

Single Page Flush

- buf/buf0flu.cc: buf_flush_write_block_low()

```
873  buf_flush_write_block_low(
874  /*======================*/
875          buf_page_t*     bpage,          /*!< in: buffer block to write */
876          buf_flush_t     flush_type,     /*!< in: type of flush */
877          bool            sync)           /*!< in: true if sync IO request */
878  {
...
918          /* Force the log to the disk before writing the modified block */
919          log_write_up_to(bpage->newest_modification, LOG_WAIT_ALL_GROUPS, TRUE);
...
939          case BUF_BLOCK_FILE_PAGE:
940                  frame = bpage->zip.data;
941                  if (!frame) {
942                          frame = ((buf_block_t*) bpage)->frame;
943                  }
944
945                  buf_flush_init_for_writing(((buf_block_t*) bpage)->frame,
946                                             bpage->zip.data
947                                             ? &bpage->zip : NULL,
948                                             bpage->newest_modification);
949                  break;
```

Flush log (transaction log – WAL)

Single Page Flush

- buf/buf0flu.cc: buf_flush_write_block_low()

```
952             if (!srv_use_doublewrite_buf || !buf_dblwr) {
953                     fil_io(OS_FILE_WRITE | OS_AIO_SIMULATED_WAKE_LATER,
954                            sync, buf_page_get_space(bpage), zip_size,
955                            buf_page_get_page_no(bpage), 0,
956                            zip_size ? zip_size : UNIV_PAGE_SIZE,
957                            frame, bpage);
958             } else if (flush_type == BUF_FLUSH_SINGLE_PAGE) {
959                     buf_dblwr_write_single_page(bpage, sync);
960             } else {
961                     ut_ad(!sync
962                     buf_dblwr_add_to_batch(bpage);
963             }
```

Doublewrite off case

Write the page to dwb, then write to datafile;
See this in dwb part

Single Page Flush

- buf/buf0flu.cc: buf_flush_write_block_low()

```
965         /* When doing single page flushing the IO is done synchronously
966         and we flush the changes to disk only for the tablespace we
967         are working on. */
968         if (sync) {
969                 ut_ad(flush_type == BUF_FLUSH_SINGLE_PAGE);
970                 fil_flush(buf_page_get_space(bpage));
971                 buf_page_io_complete(bpage);
972         }
973
974         /* Increment the counter of I/
975         for selecting LRU policy. */
976         buf_LRU_stat_inc_io();
977 }
```

Sync buffered write to disk; call **fsync** by ***fil_flush()***

FLUSH A PAGE

PART 2 : BATCH FLUSH

Batch Flush

- Background flusher (= page cleaner thread)
  - Independent thread for flushing a dirty pages from buffer pools to storage
  - Regularly (per 1000ms) do **flush from LRU** tail or
  - Do **flush by dirty page percent** (configurable)

- Thread definition
  - buf/buf0flu.cc: *DECLARE_THREAD(buf_flush_page_cleaner_thread)*

Background Flusher

- buf/buf0flu.cc: DECLARE_THREAD(buf_flush_page_cleaner_thread)

```
2385  DECLARE_THREAD(buf_flush_page_cleaner_thread)(
2386  /*=======================================*/
2387          void*   arg __attribute__((unused)))
2388                          /*!< in: a dummy parameter required by
2389                          os_thread_create */
2390  {
...
2408          while (srv_shutdown_state == SRV_SHUTDOWN_NONE) {
2409
2410                  /* The page_cleaner skips sleep if the server is
2411                  idle and there are no pending IOs in the buffer pool
2412                  and there is work to do. */
2413                  if (srv_check_activity(last_activity)
2414                      || buf_get_n_pending_read_ios()
2415                      || n_flushed == 0) {
2416                          page_cleaner_sleep_if_needed(next_loop_time);
2417                  }
2418
2419                  next_loop_time = ut_time_ms() + 1000;
```

Run until shutdown

Background Flusher

- buf/buf0flu.cc: DECLARE_THREAD(buf_flush_page_cleaner_thread)

```
2421    if (srv_check_activity(last_activity)) {
2422            last_activity = srv_get_activity_count();
2423
2424            /* Flush pages from end of LRU if required */
2425            n_flushed = buf_flush_LRU_tail();
2426
2427            /* Flush pages from flush_list if required */
2428            n_flushed += page_cleaner_flush_pages_if_needed();
2429    } else {
2430            n_flushed = page_cleaner_do_flush_batch(
2431                                            PCT_IO(100),
2432                                            LSN_MAX);
2433
2434            if (n_flushed) {
2435                    MONITOR_INC_VALUE_CUMULATIVE(
2436                            MONITOR_FLUSH_BACKGROUND_TOTAL_PAGE,
2437                            MONITOR_FLUSH_BACKGROUND_COUNT,
2438                            MONITOR_FLUSH_BACKGROUND_PAGES,
2439                            n_flushed);
2440            }
2441    }
```

Something has been changed!

Nothing has been changed

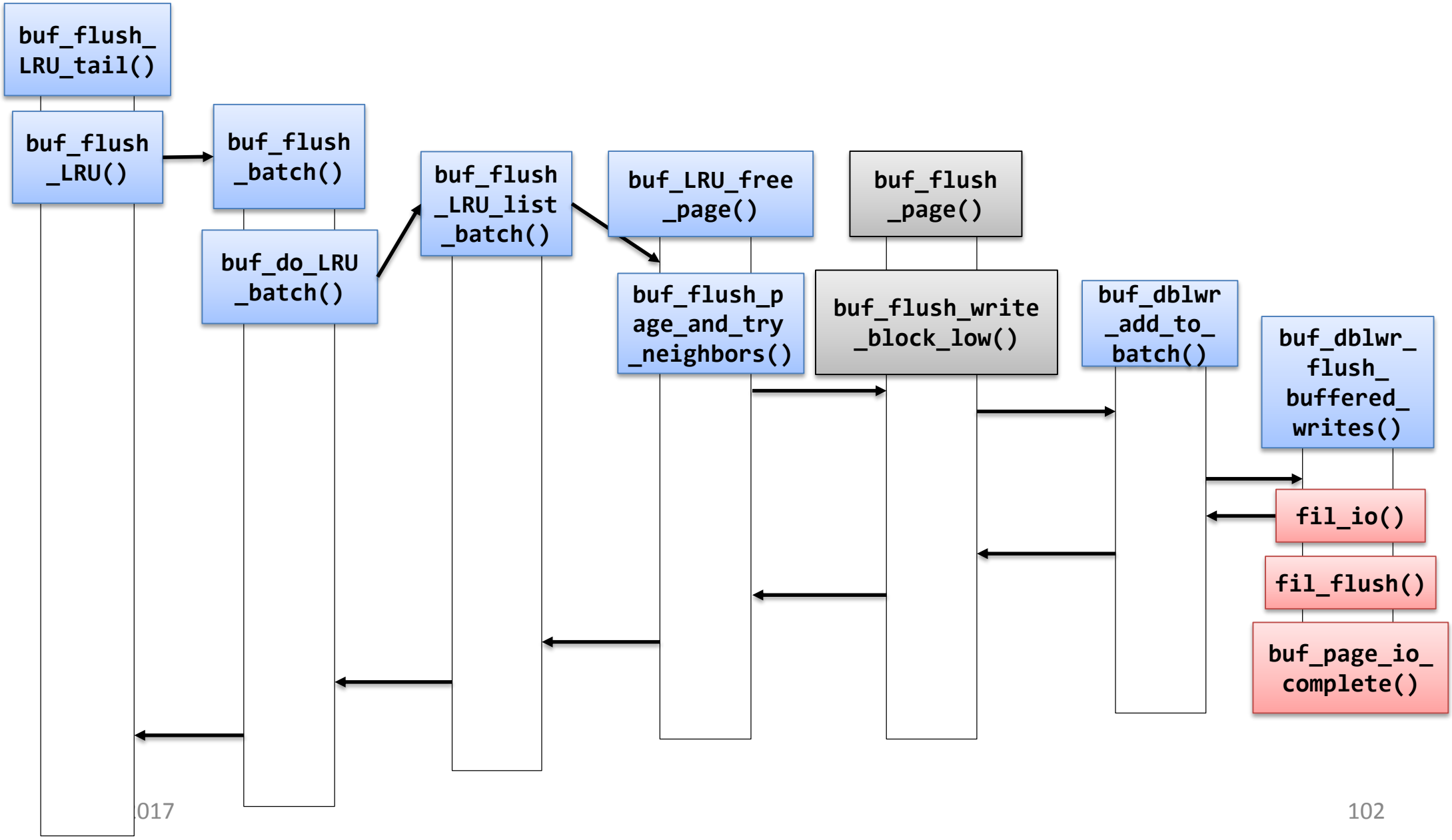
LRU List Batch Flush

- buf/buf0flu.cc: *buf_flush_LRU_tail()*

- Clears up tail of the LRU lists:
  - Put replaceable pages at the tail of LRU to the free list
  - Flush dirty pages at the tail of LRU to the disk

- *srv_LRU_scan_depth*
  - Scan each buffer pool at this amount
  - Configuable: *innodb_LRU_scan_depth*

LRU List Batch Flush



LRU List Batch Flush

- buf/buf0flu.cc: buf_flush_LRU_tail()

```
2065  buf_flush_LRU_tail(void)
2066  /*=====================*/
2067  {
2068          ulint   total_flushed = 0;
2069
2070          for (ulint i = 0; i < srv_buf_pool_instances; i++) {
2071
2072                  buf_pool_t*     buf_pool = buf_pool_from_array(i);
2073                  ulint           scan_depth;
2074
2075                  /* srv_LRU_scan_depth can be arbitrarily large value.
2076                  We cap it with current LRU size. */
2077                  buf_pool_mutex_enter(buf_pool);
2078                  scan_depth = UT_LIST_GET_LEN(buf_pool->LRU);
2079                  buf_pool_mutex_exit(buf_pool);
2080
2081                  scan_depth = ut_min(srv_LRU_scan_depth, scan_depth);
```

Per buffer pool instance

LRU List Batch Flush

- buf/buf0flu.cc: buf_flush_LRU_tail()

```
2086                for (ulint j = 0;
2087                     j < scan_depth;
2088                     j += PAGE_CLEANER_LRU_BATCH_CHUNK_SIZE) {
2089
2090                        ulint   n_flushed = 0;
2091
2092                        /* Currently page_cleaner is the only thread
2093                        that can trigger an LRU flush. It is possible
2094                        that a batch triggered during last iteration is
2095                        still running, */
2096                        if (buf_flush_LRU(buf_pool,
2097                                          PAGE_CLEANER_LRU_BATCH_CHUNK_SIZE,
2098                                          &n_flushed)) {
2099
2100                                /* Allowed only one batch per
2101                                buffer pool instance. */
2102                                buf_flush_wait_batch_end(
2103                                        buf_pool, BUF_FLUSH_LRU);
2104                        }
```

Chunk size = 100

LRU List Batch Flush

- buf/buf0flu.cc: buf_flush_LRU()

```
1840  buf_flush_LRU(
1841  /*==========*/
1842          buf_pool_t*     buf_pool,       /*!< in/out: buffer pool instance */
1843          ulint           min_n,          /*!< in: wished minimum mumber of blocks
1844                                          flushed (it is not guaranteed that the
1845                                          actual number is that big, though) */
1846          ulint*          n_processed)    /*!< out: the number of pages
1847                                          which were processed is passed
1848                                          back to caller. Ignored if NULL */
1849  {
 ...
1860          page_count = buf_flush_batch(buf_pool, BUF_FLUSH_LRU, min_n, 0);
1861
1862          buf_flush_end(buf_pool, BUF_FLUSH_LRU);
1863
1864          buf_flush_common(BUF_FLUSH_LRU, page_count);
```

Batch LRU flush

LRU List Batch Flush

- buf/buf0flu.cc: buf_flush_batch()

```
1691            buf_pool_mutex_enter(buf_pool);
1692
1693            /* Note: The buffer pool mutex is released and reacquired within
1694            the flush functions. */
1695            switch (flush_type) {
1696            case BUF_FLUSH_LRU:
1697                    count = buf_do_LRU_batch(buf_pool, min_n);
1698                    break;
1699            case BUF_FLUSH_LIST:
1700                    count = buf_do_flush_list_batch(buf_pool, min_n, lsn_limit);
1701                    break;
1702            default:
1703                    ut_error;
1704            }
1705
1706            buf_pool_mutex_exit(buf_pool);
```

Do LRU batch

LRU List Batch Flush

- buf/buf0flu.cc: buf_do_LRU_batch()

```
1551  buf_do_LRU_batch(
1552  /*=============*/
1553          buf_pool_t*     buf_pool,       /*!< in: buffer pool instance */
1554          ulint           max)            /*!< in: desired number of
1555                                          blocks in the free_list */
1556  {
1557          ulint   count = 0;
1558
1559          if (buf_LRU_evict_from_unzip_LRU(buf_pool)) {
1560                  count += buf_free_from_unzip_LRU_list_batch(buf_pool, max);
1561          }
1562
1563          if (max > count) {
1564                  count += buf_flush_LRU_list_batch(buf_pool, max - count);
1565          }
1566
1567          return(count);
1568  }
```

LRU List Batch Flush

- buf/buf0flu.cc: buf_flush_LRU_list_batch()

```
1434  buf_flush_LRU_list_batch(
1435  /*=====================*/
1436          buf_pool_t*     buf_pool,       /*!< in: buffer pool instance */
1437          ulint           max)            /*!< in: desired number of
1438                                          blocks in the free_list */
1439  {
1440          buf_page_t*     bpage;
1441          ulint           count = 0;
1442          ulint           scanned = 0;
1443          ulint           free_len = UT_LIST_GET_LEN(buf_pool->free);
1444          ulint           lru_len = UT_LIST_GET_LEN(buf_pool->LRU);
1445
1446          ut_ad(buf_pool_mutex_own(buf_pool));
1447
1448          bpage = UT_LIST_GET_LAST(buf_pool->LRU);
```

Get the last page from LRU

LRU List Batch Flush

- buf/buf0flu.cc: buf_flush_LRU_list_batch()

```
1449            while (bpage != NULL && count < max
1450                   && free_len < srv_LRU_scan_depth
1451                   && lru_len > BUF_LRU_MIN_LEN) {
1452
1453                    ib_mutex_t* block_mutex = buf_page_get_mutex(bpage);
1454                    ibool    evict;
1455
1456                    mutex_enter(block_mutex);
1457                    evict = buf_flush_ready_for_replace(bpage);
1458                    mutex_exit(block_mutex);
1459
1460                    ++scanned;
```

LRU List Batch Flush

- buf/buf0flu.cc: buf_flush_ready_for_replace()

```
497  buf_flush_ready_for_replace(
498  /*========================*/
499          buf_page_t*     bpage)  /*!< in: buffer control block, must be
500                                  buf_page_in_file(bpage) and in the LRU list */
501  {
502  #ifdef UNIV_DEBUG
503          buf_pool_t*     buf_pool = buf_pool_from_bpage(bpage);
504          ut_ad(buf_pool_mutex_own(buf_pool));
505  #endif /* UNIV_DEBUG */
506          ut_ad(mutex_own(buf_page_get_mutex(bpage)));
507          ut_ad(bpage->in_LRU_list);
508
509          if (buf_page_in_file(bpage)) {
510
511                  return(bpage->oldest_modification == 0
512                         && bpage->buf_fix_count == 0
513                         && buf_page_get_io_fix(bpage) == BUF_IO_NONE);
514          }
```

Check whether current page is **clean page** or not

LRU List Batch Flush

- buf/buf0flu.cc: buf_flush_LRU_list_batch()

```
1473                    if (evict) {
1474                            if (buf_LRU_free_page(bpage, true)) {
1475                                    /* buf_pool->mutex was potentially
1476                                    released and reacquired. */
1477                                    bpage = UT_LIST_GET_LAST(buf_pool->LRU);
1478                            } else {
1479                                    bpage = UT_LIST_GET_PREV(LRU, bpage);
1480                            }
1481                    } else {
```

It there is any replaceable page, free the page

LRU List Batch Flush

- buf/buf0flu.cc: buf_flush_LRU_list_batch()

```
1481                } else {
1482                        ulint           space;
1483                        ulint           offset;
1484                        buf_page_t*     prev_bpage;
1485
1486                        prev_bpage = UT_LIST_GET_PREV(LRU, bpage);
1487
1488                        /* Save the previous bpage */
1489
1490                        if (prev_bpage != NULL) {
1491                                space = prev_bpage->space;
1492                                offset = prev_bpage->offset;
1493                        } else {
1494                                space = ULINT_UNDEFINED;
1495                                offset = ULINT_UNDEFINED;
1496                        }
1497
1498                        if (!buf_flush_page_and_try_neighbors(
1499                                bpage, BUF_FLUSH_LRU, max, &count)) {
1500
1501                                bpage = prev_bpage;
```

Else, try to flush neighbor pages

LRU List Batch Flush

- buf/buf0flu.cc: buf_flush_page_and_try_neighbors()

```
1265                    if (flush_type != BUF_FLUSH_LRU
1266                        || i == offset
1267                        || buf_page_is_old(bpage)) {
1268
1269                            ib_mutex_t* block_mutex = buf_page_get_mutex(bpage);
1270
1271                            mutex_enter(block_mutex);
1272
1273                            if (buf_flush_ready_for_flush(bpage, flush_type)
1274                                && (i == offset || bpage->buf_fix_count == 0)
1275                                && buf_flush_page(
1276                                        buf_pool, bpage, flush_type, false)) {
1277
1278                                    ++count;
1279
1280                                    continue;
1281                            }
1282
1283                            mutex_exit(block_mutex);
1284                    }
```

Flush page, but no sync

LRU List Batch Flush

- buf/buf0flu.cc: buf_flush_write_block_low()

```
952             if (!srv_use_doublewrite_buf || !buf_dblwr) {
953                     fil_io(OS_FILE_WRITE | OS_AIO_SIMULATED_WAKE_LATER,
954                            sync, buf_page_get_space(bpage), zip_size,
955                            buf_page_get_page_no(bpage), 0,
956                            zip_size ? zip_size : UNIV_PAGE_SIZE,
957                            frame, bpage);
958             } else if (flush_type == BUF_FLUSH_SINGLE_PAGE) {
959                     buf_dblwr_write_single_page(bpage, sync);
960             } else {
961                     ut_ad(!sync);
962                     buf_dblwr_add_to_batch(bpage);
963             }
```

Doublewrite off case

Add the page to the dwb buffer;
See this in dwb part

LRU List Batch Flush

- Now, victim pages are gathered for replacement
- We need to flush them to disk

- We can do this by calling *buf_flush_common()*

LRU List Batch Flush

- buf/buf0flu.cc: buf_flush_LRU()

```
1840  buf_flush_LRU(
1841  /*==========*/
1842          buf_pool_t*     buf_pool,       /*!< in/out: buffer pool instance */
1843          ulint           min_n,          /*!< in: wished minimum mumber of blocks
1844                                          flushed (it is not guaranteed that the
1845                                          actual number is that big, though) */
1846          ulint*          n_processed)    /*!< out: the number of pages
1847                                          which were processed is passed
1848                                          back to caller. Ignored if NULL */
1849  {
 ...
1860          page_count = buf_flush_batch(buf_pool, BUF_FLUSH_LRU, min_n, 0);
1861
1862          buf_flush_end(buf_pool, BUF_FLUSH_LRU);
1863
1864          buf_flush_common(BUF_FLUSH_LRU, page_count);
```

LRU List Batch Flush

- buf/buf0flu.cc: buf_flush_common()

```
1724  buf_flush_common(
1725  /*=============*/
1726          buf_flush_t     flush_type,     /*!< in: type of flush */
1727          ulint           page_count)     /*!< in: number of pages flushed */
1728  {
1729          buf_dblwr_flush_buffered_writes();
1730
1731          ut_a(flush_type == BUF_FLUSH_LRU ||
1732
1733  #ifdef UNIV_DEBUG
1734          if (buf_debug_prints && page_count > 0) {
1735                  fprintf(stderr, flush_type == BUF_FLUSH_LRU
1736                          ? "Flushed %lu pages in LRU flush\n"
1737                          : "Flushed %lu pages in flush list flush\n",
1738                          (ulong) page_count);
1739          }
1740  #endif /* UNIV_DEBUG */
1741
1742          srv_stats.buf_pool_flushed.add(page_count);
1743  }
```

- **flush** all pages we gathered so far
- **write** the pages to **dwb** area first
- then **issue** it to **datafile**

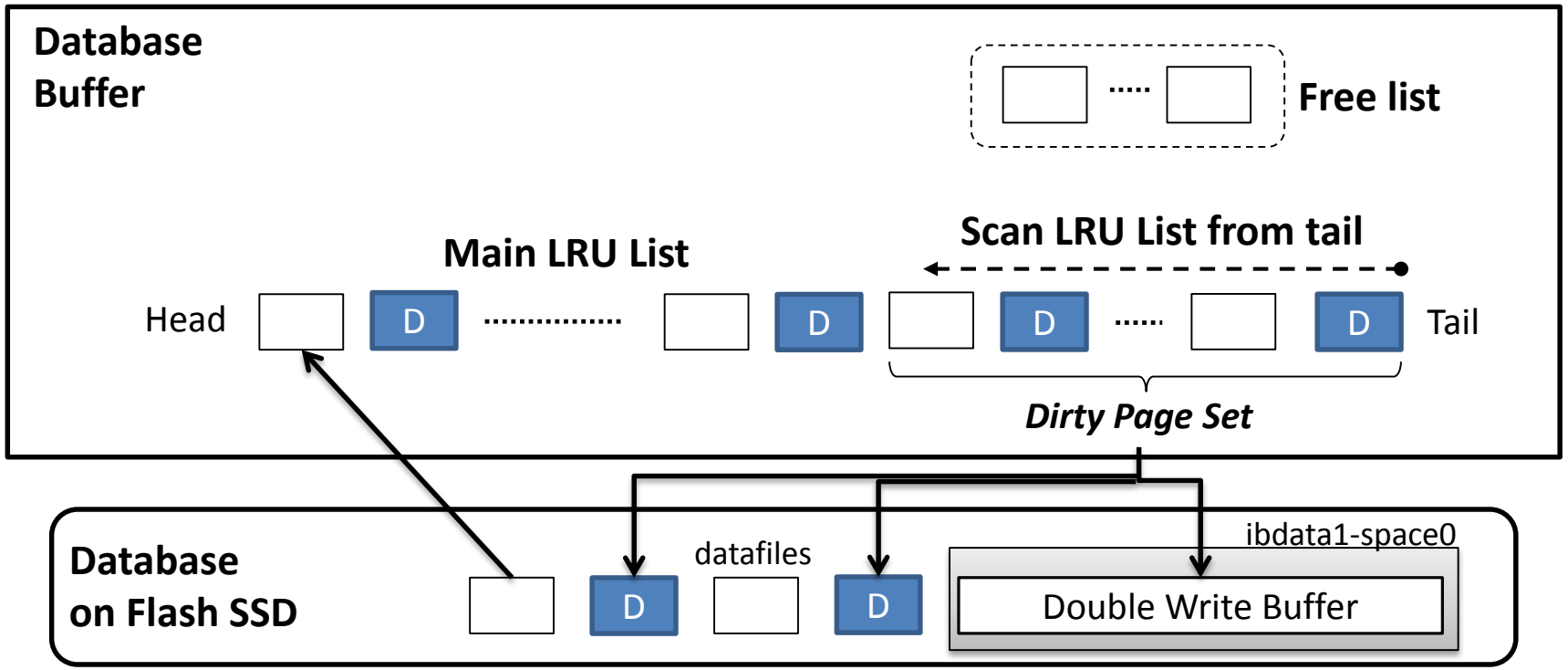
; See this later

DOUBLEWRITE BUFFER

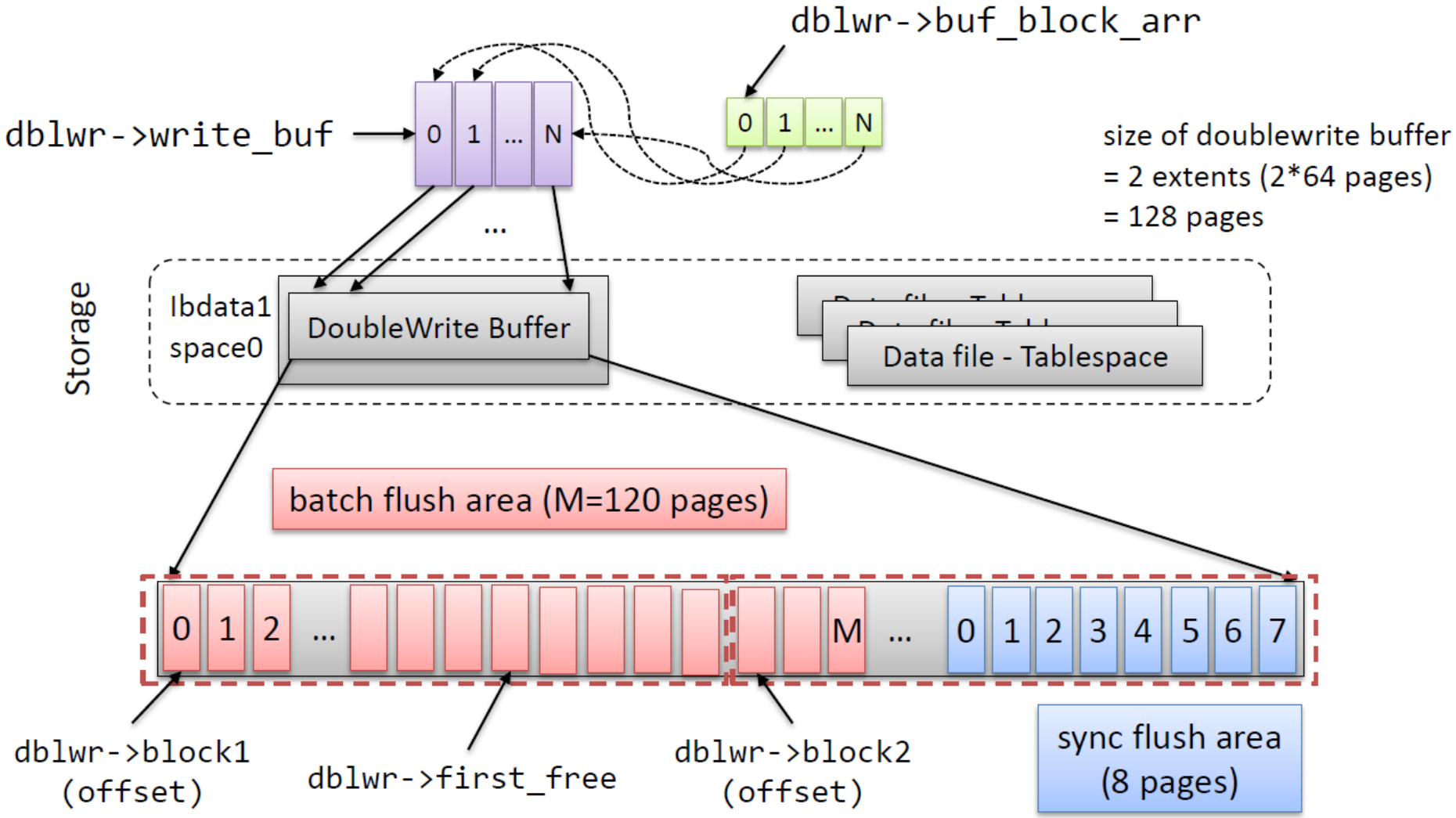
PART 1 : ARCHITECTURE

Double Write Buffer

- To avoid torn page (partial page) written problem
- Write **dirty** pages to *special storage area* in **system tablespace** priori to write database file



DWB Architecture



DWB Struct

- include/buf0dblwr.cc: buf_dblwr_t

```
126  struct buf_dblwr_t{
127          ib_mutex_t      mutex;  /*!< mutex protecting the first_free
128                                  field and write_buf */
129          ulint           block1; /*!< the page number of the first
130                                  doublewrite block (64 pages) */
131          ulint           block2; /*!< page number of the second block */
132          ulint           first_free;/*!< first free position in write_buf
133                                  measured in units of UNIV_PAGE_SIZE */
134          ulint           b_reserved;/*!< number of slots currently reserved
135                                  for batch flush. */
136          os_event_t      b_event;/*!< event where threads wait for a
137                                  batch flush to end. */
138          ulint           s_reserved;/*!< number of slots currently
139                                  reserved for single page flushes. */
```

DWB Struct

- include/buf0dblwr.cc: buf_dblwr_t

```
140         os_event_t      s_event;/*!< event where threads wait for a
141                                 single page flush slot. */
142         bool*           in_use; /*!< flag used to indicate if a slot is
143                                 in use. Only used for single page
144                                 flushes. */
145         bool            batch_running;/*!< set to TRUE if currently a batch
146                                 is being written from the doublewrite
147                                 buffer. */
148         byte*           write_buf;/*!< write buffer used in writing to the
149                                 doublewrite buffer, aligned to an
150                                 address divisible by UNIV_PAGE_SIZE
151                                 (which is required by Windows aio) */
152         byte*           write_buf_unaligned;/*!< pointer to write_buf,
153                                 but unaligned */
154         buf_page_t**    buf_block_arr;/*!< array to store pointers to
155                                 the buffer blocks which have been
156                                 cached to write_buf */
```

DOUBLEWRITE BUFFER

PART 2 : SINGLE PAGE FLUSH

Single Page Flush

- buf/buf0flu.cc: buf_flush_write_block_low()

```
952             if (!srv_use_doublewrite_buf || !buf_dblwr) {
953                     fil_io(OS_FILE_WRITE | OS_AIO_SIMULATED_WAKE_LATER,
954                            sync, buf_page_get_space(bpage), zip_size,
955                            buf_page_get_page_no(bpage), 0,
956                            zip_size ? zip_size : UNIV_PAGE_SIZE,
957                            frame, bpage);
958             } else if (flush_type == BUF_FLUSH_SINGLE_PAGE) {
959                     buf_dblwr_write_single_page(bpage, sync);
960             } else {
961                     ut_ad(!sync);
962                     buf_dblwr_add_to_batch(bpage);
963             }
```

Single Page Flush

- buf/buf0dblwr.cc: buf_dblwr_write_single_page()

```
1055  buf_dblwr_write_single_page(
1056  /*========================*/
1057          buf_page_t*     bpage,  /*!< in: buffer block to write */
1058          bool            sync)   /*!< in: true if sync IO requested */
1059  {
 ...
1073          size = 2 * TRX_SYS_DOUBLEWRITE_BLOCK_SIZE;
1074          ut_a(size > srv_doublewrite_batch_size);
1075          n_slots = size - srv_doublewrite_batch_size;
```

# of slots for single page flush
= 2 * DOUBLEWRITE_BLOCK_SIZE – BATCH_SIZE
= 128 – 120
= **8**

Single Page Flush

- buf/buf0dblwr.cc: buf_dblwr_write_single_page()

```
1091  retry:
1092          mutex_enter(&buf_dblwr->mutex);
1093          if (buf_dblwr->s_reserved == n_slots) {
1094
1095                  /* All slots are reserved. */
1096                  ib_int64_t      sig_count =
1097                          os_event_reset(buf_dblwr->s_event);
1098                  mutex_exit(&buf_dblwr->mutex);
1099                  os_event_wait_low(buf_dblwr->s_event, sig_count);
1100
1101                  goto retry;
1102          }
1103
1104          for (i = srv_doublewrite_batch_size; i < size; ++i) {
1105
1106                  if (!buf_dblwr->in_use[i]) {
1107                          break;
1108                  }
1109          }
```

If all slots are reserved, wait until current dblwr done

Find a free slot

Single Page Flush

- buf/buf0dblwr.cc: buf_dblwr_write_single_page()

```
1155                    /* It is a regular page. Write it directly to the
1156                    doublewrite buffer */
1157                    fil_io(OS_FILE_WRITE, true, TRX_SYS_SPACE, 0,
1158                           offset, 0, UNIV_PAGE_SIZE,
1159                           (void*) ((buf_block_t*) bpage)->frame,
1160                           NULL);
1161            }
1162
1163            /* Now flush the doublewrit
1164            fil_flush(TRX_SYS_SPACE);
1165
1166            /* We know that the write has been flushed to disk now
1167            and during recovery we will find it in the doublewrite buffer
1168            blocks. Next do the write to the intended positi
1169            buf_dblwr_write_block_to_datafile(bpage, sync);
1170    }
```

Write block to dwb area in disk (**synchronous**)

Sync system tablespace for dwb area in disk

Write to datafile (**synchronous**)

Single Page Flush

- buf/buf0dblwr.cc: buf_dblwr_write_block_to_datafile()

```
782  buf_dblwr_write_block_to_datafile(
783  /*==============================*/
784          const buf_page_t*       bpage,  /*!< in: page to write */
785          bool                    sync)   /*!< in: true if sync IO
786                                          is requested */
787  {
...
807          const buf_block_t* block = (buf_block_t*) bpage;
808          ut_a(buf_block_get_state(block) == BUF_BLOCK_FILE_PAGE);
809          buf_dblwr_check_page_lsn(block->frame);
810
811          fil_io(flags, sync, buf_block_get_space(block), 0,
812                 buf_block_get_page_no(block), 0, UNIV_PAGE_SIZE,
813                 (void*) block->frame, (void*) block);
814
815  }
```

In single page flush, *sync* = **true**

Issue write operation to datafile

Single Page Flush

- buf/buf0flu.cc: buf_flush_write_block_low()

```
965             /* When doing single page flushing the IO is done synchronously
966             and we flush the changes to disk only for the tablespace we
967             are working on. */
968             if (sync) {
969                     ut_ad(flush_type == BUF_FLUSH_SINGLE_PAGE);
970                     fil_flush(buf_page_get_space(bpage));
971                     buf_page_io_complete(bpage);
972             }
973
974             /* Increment the counter of I/
975             for selecting LRU policy. */
976             buf_LRU_stat_inc_io();
977     }
```

Sync buffered write to disk; call **fsync** by ***fil_flush()***

Single Page Flush

- buf/buf0buf.cc: buf_page_io_complete()

```
4162  buf_page_io_complete(
4163  /*=================*/
4164          buf_page_t*     bpage)  /*!< in: pointer to the block in question */
4165  {
4166          enum buf_io_fix io_type;
4167          buf_pool_t*     buf_pool = buf_pool_from_bpage(bpage);
4168          const ibool     uncompressed = (buf_page_get_state(bpage)
4169                                          == BUF_BLOCK_FILE_PAGE);
4170
4171          ut_a(buf_page_in_file(bpage));
4172
4173          /* We do not need protect io_fix here by mutex to read
4174          it because this is the only function where we can change the value
4175          from BUF_IO_READ or BUF_IO_WRITE to some other value, and our code
4176          ensures that this is the only thread that handles the i/o for this
4177          block. */
4178
4179          io_type = buf_page_get_io_fix(bpage);
```

Get io type (In this case, BUF_IO_WRITE)

Single Page Flush

- buf/buf0buf.cc: buf_page_io_complete()

```
4329            buf_pool_mutex_enter(buf_pool);
4330            mutex_enter(buf_page_get_mutex(bpage));
4331
4332    #ifdef UNIV_IBUF_COUNT_DEBUG
4333            if (io_type == BUF_IO_WRITE || uncompressed) {
4334                    /* For BUF_IO_READ of compressed-only blocks, the
4335                    buffered operations will be merged by buf_page_get_gen()
4336                    after the block has been uncompressed. */
4337                    ut_a(ibuf_count_get(bpage->space, bpage->offset) == 0);
4338            }
4339    #endif
4340            /* Because this thread which does the unlocking is not the same that
4341            did the locking, we use a pass value != 0 in unlock, which simply
4342            removes the newest lock debug record, without checking the thread
4343            id. */
4344
4345            buf_page_set_io_fix(bpage, BUF_IO_NONE);
```

Set io fix to BUF_IO_NONE

Single Page Flush

- buf/buf0buf.cc: buf_page_io_complete()

```
4482            case BUF_IO_WRITE:
4483                    /* Write means a flush operation: call the completion
4484                    routine in the flush system */
4485
4486                    buf_flush_write_complete(bpage);
4487
4488                    if (uncompressed) {
4489                            rw_lock_s_unlock_gen(&((buf_block_t*) bpage)->lock,
4490                                                 BUF_IO_WRITE);
4491                    }
4492
4493                    buf_pool->stat.n_pages_written++;
4494
4495                    break;
```

Single Page Flush

- buf/buf0flu.cc: buf_flush_write_complete()

```
712  buf_flush_write_complete(
713  /*=====================*/
714          buf_page_t*     bpage)  /*!< in: pointer to the block in question */
715  {
716          buf_flush_t     flush_type;
717          buf_pool_t*     buf_pool = buf_pool_from_bpage(bpage);
718
719          ut_ad(bpage);
720
721          buf_flush_remove(bpage);
722
723          flush_type = buf_page_get_flush_type(bpage);
724          buf_pool->n_flush[flush_type]--;
 …
737          buf_dblwr_update(bpage, flush_type);
738  }
```

Remove the block from the flush list

Single Page Flush

- buf/buf0dblwr.cc: buf_dblwr_update()

```
667         case BUF_FLUSH_SINGLE_PAGE:
668                 {
669                         const ulint size = 2 * TRX_SYS_DOUBLEWRITE_BLOCK_SIZE;
670                         ulint i;
671                         mutex_enter(&buf_dblwr->mutex);
672                         for (i = srv_doublewrite_batch_size; i < size; ++i) {
673                                 if (buf_dblwr->buf_block_arr[i] == bpage) {
674                                         buf_dblwr->s_reserved--;
675                                         buf_dblwr->buf_block_arr[i] = NULL;
676                                         buf_dblwr->in_use[i] = false;
677                                         break;
678                                 }
679                         }
680
681                         /* The block we are looking for must exist as a
682                         reserved block. */
683                         ut_a(i < size);
684                 }
685                 os_event_set(buf_dblwr->s_event);
686                 mutex_exit(&buf_dblwr->mutex);
687                 break;
```

Free the dwb slot of the target page

Single Page Flush

- Single page flush is performed in the context of the query thread itself
- Single page flush mode iterates over the LRU list of a buffer pool instance, while holding the buffer pool mutex
- It might have trouble in getting a free doublewrite buffer slot (total **8** slots)
- **In result, it makes the overall performance worse**

DOUBLEWRITE BUFFER

PART 3 : BATCH FLUSH

Batch Flush

- buf/buf0flu.cc: buf_flush_write_block_low()

```
952             if (!srv_use_doublewrite_buf || !buf_dblwr) {
953                     fil_io(OS_FILE_WRITE | OS_AIO_SIMULATED_WAKE_LATER,
954                            sync, buf_page_get_space(bpage), zip_size,
955                            buf_page_get_page_no(bpage), 0,
956                            zip_size ? zip_size : UNIV_PAGE_SIZE,
957                            frame, bpage);
958             } else if (flush_type == BUF_FLUSH_SINGLE_PAGE) {
959                     buf_dblwr_write_single_page(bpage, sync);
960             } else {
961                     ut_ad(!sync);
962                     buf_dblwr_add_to_batch(bpage);
963             }
```

Batch Flush

- buf/buf0dblwr.cc: buf_dblwr_add_to_batch()

```
976  try_again:
977          mutex_enter(&buf_dblwr->mutex);
978
979          ut_a(buf_dblwr->first_free <= srv_doublewr
980
981          if (buf_dblwr->batch_running) {
982
983                  /* This not nearly as bad as it looks. There is only
984                  page_cleaner thread which does background flushing
985                  in batches therefore it is unlikely to be a contention
986                  point. The only exception is when a user thread is
987                  forced to do a flush batch because of a sync
988                  checkpoint. */
989                  ib_int64_t      sig_count = os_event_reset(buf_dblwr->b_event);
990                  mutex_exit(&buf_dblwr->mutex);
991
992                  os_event_wait_low(buf_dblwr->b_event, sig_count);
993                  goto try_again;
994          }
```

If another batch is already running, wait until done

Batch Flush

- buf/buf0dblwr.cc: buf_dblwr_add_to_batch()

```
996             if (buf_dblwr->first_free == srv_doublewrite_batch_size) {
997                     mutex_exit(&(buf_dblwr->mutex));
998
999                     buf_dblwr_flush_buffered_writes();
1000
1001                    goto try_again;
1002            }
```

If all slots for batch flush in dwb buffer is reserved, flush dwb buffer

Batch Flush

- buf/buf0dblwr.cc: buf_dblwr_add_to_batch()

```
1016                    ut_a(buf_page_get_state(bpage) == BUF_BLOCK_FILE_PAGE);
1017                    UNIV_MEM_ASSERT_RW(((buf_block_t*) bpage)->frame,
1018                                       UNIV_PAGE_SIZE);
1019
1020                    memcpy(buf_dblwr->write_buf
1021                           + UNIV_PAGE_SIZE * buf_dblwr->first_free,
1022                           ((buf_block_t*) bpage)->frame, UNIV_PAGE_SIZE);
1023            }
1024
1025            buf_dblwr->buf_block_arr[buf_dblwr->first_free] = bpage;
1026
1027            buf_dblwr->first_free++;
1028            buf_dblwr->b_reserved++;
```

After flushing, copy current block to buf_dblwr->first_free

Batch Flush

- buf/buf0dblwr.cc: buf_dblwr_flush_buffered_writes()

```
825  buf_dblwr_flush_buffered_writes(void)
826  /*=====================================*/
827  {
828          byte*           write_buf;
829          ulint           first_free;
830          ulint           len;
831
832          if (!srv_use_doublewrite_buf || buf_dblwr == NULL) {
833                  /* Sync the writes to the disk. */
834                  buf_dblwr_sync_datafiles();
835                  return;
836          }
837
```

Doublewrite off case

Batch Flush

- buf/buf0dblwr.cc: buf_dblwr_sync_datafiles()

```
107  buf_dblwr_sync_datafiles()
108  /*======================*/
109  {
110          /* Wake possible simulated aio thread to actually post the
111          writes to the operating system */
112          os_aio_simulated_wake_handler_threads();
113
114          /* Wait that all async writes to tablespaces have been posted to
115          the OS */
116          os_aio_wait_until_no_pending_writes();
117
118          /* Now we flush the data to disk (for ex                    */
119          fil_flush_file_spaces(FIL_TABLESPACE);
120  }
```

- ➔ fil_flush()
- ➔ os_file_flush()
- ➔ os_file_fsync()
- ➔ fsync()

Batch Flush

- buf/buf0dblwr.cc: buf_dblwr_flush_buffered_writes()

```
865         /* Disallow anyone else to post to doublewrite buffer or to
866         start another batch of flushing. */
867         buf_dblwr->batch_running = true;
868         first_free = buf_dblwr->first_free;
869
870         /* Now safe to release the mutex. Note that though no other
871         thread is allowed to post to the doublewrite batch flushing
872         but any threads working on single page flushes are allowed
873         to proceed. */
874         mutex_exit(&buf_dblwr->mutex);
875
876         write_buf = buf_dblwr->write_buf;
```

Change batch running status

Exit mutex

Nobody won't be here except me!

Batch Flush

- buf/buf0dblwr.cc: buf_dblwr_flush_buffered_writes()

```
902         /* Write out the first block of the doublewrite buffer */
903         len = ut_min(TRX_SYS_DOUBLEWRITE_BLOCK_SIZE,
904                      buf_dblwr->first_free) * UNIV_PAGE_SIZE;
905
906         fil_io(OS_FILE_WRITE, true, TRX_SYS_SPACE, 0,
907                buf_dblwr->block1, 0, len,
908                (void*) write_buf, NULL);
909
910         if (buf_dblwr->first_free <= TRX_SYS_DOUBLEWRITE_BLOCK_SIZE) {
911                 /* No unwritten pages in the second block. */
912                 goto flush;
913         }
```

Issue write op for block1 (**synchronous**)

If current write uses only block1, then flush

Batch Flush

- buf/buf0dblwr.cc: buf_dblwr_flush_buffered_writes()

```
915         /* Write out the second block of the doublewrite buffer. */
916         len = (buf_dblwr->first_free - TRX_SYS_DOUBLEWRITE_BLOCK_SIZE)
917               * UNIV_PAGE_SIZE;
918
919         write_buf = buf_dblwr->write_buf
920                     + TRX_SYS_DOUBLEWRITE_BLOCK_SIZE * UNIV_PAGE_SIZE;
921
922         fil_io(OS_FILE_WRITE, true, TRX_SYS_SPACE, 0,
923                buf_dblwr->block2, 0, len,
924                (void*) write_buf, NULL);
```

Issue write op for block2 (**synchronous**)

Batch Flush

- buf/buf0dblwr.cc: buf_dblwr_flush_buffered_writes()

```
926  flush:
927          /* increment the doublewrite flushed pages counter */
928          srv_stats.dblwr_pages_written.add(buf_dblwr->first_free);
929          srv_stats.dblwr_writes.inc();
930
931          /* Now flush the doublewrite buffer data to disk */
932          fil_flush(TRX_SYS_SPACE);
...
950          for (ulint i = 0; i < first_free; i++) {
951                  buf_dblwr_write_block_to_datafile(
952                          buf_dblwr->buf_block_arr[i], false);
953          }
954
955          /* Wake possible simulated aio
956          writes to the operating system. We don't flush the files
957          at this point. We leave it to the IO helper thread to flush
958          datafiles when the whole batch has been processed. */
959          os_aio_simulated_wake_handler_threads();
960  }
```

Flush (fsync) system table space

Write all blocks to datafile (**asynchronous**)

Batch Flush

- After submitting aio requests,
  - fil_aio_wait()
  - buf_page_io_complete()
  - buf_flush_write_complete()
  - buf_dblwr_update()

Batch Flush

- buf/buf0dblwr.cc: buf_dblwr_update()

```
643             case BUF_FLUSH_LRU:
644                     mutex_enter(&buf_dblwr->mutex);
650                     buf_dblwr->b_reserved--;
651
652                     if (buf_dblwr->b_reserved == 0) {
653                             mutex_exit(&buf_dblwr->mutex);
654                             /* This will finish the batch. Sync data files
655                             to the disk. */
656                             fil_flush_file_spaces(FIL_TABLESPACE);
657                             mutex_enter(&buf_dblwr->mutex);
658
659                             /* We can now reuse the doublewrite memory buffer: */
660                             buf_dblwr->first_free = 0;
661                             buf_dblwr->batch_running = false;
662                             os_event_set(buf_dblwr->b_event);
663                     }
664
665                     mutex_exit(&buf_dblwr->mutex);
666                     break;
```

Flush datafile

Reset dwb

SYNCHRONIZATION

InnoDB Synchronization

- InnoDB implements its own mutexes & RW-locks for buffer management

- **Latch** in InnoDB
  - A lightweight structure used by InnoDB to implement a lock
  - Typically held for a brief time (milliseconds or microseconds)
  - A general term that includes both **mutexes** (for **exclusive** access) and **rw-locks** (for **shared** access)

Mutex in InnoDB

- The low-level object to represent and enforce **exclusive-access locks** to internal in-memory data structures

- Once the lock is acquired, any other process, thread, and so on is prevented from acquiring the same lock

Mutex in InnoDB

- Example code in InnoDB
  - buf/buf0buf.cc: buf_wait_for_read()

```
2696                    mutex_enter(mutex);
2697
2698                    io_fix = buf_block_get_io_fix(block);
2699
2700                    mutex_exit(mutex);
```

Get current IO fix of the block

Mutex in InnoDB

- Example code in InnoDB
  - buf/buf0flu.cc: buf_flush_batch()

```
1691        buf_pool_mutex_enter(buf_pool);
1692
1693        /* Note: The buffer pool mutex is released and reacquired within
1694        the flush functions. */
1695        switch (flush_type) {
1696        case BUF_FLUSH_LRU:
1697                count = buf_do_LRU_batch(buf_pool, min_n);
1698                break;
1699        case BUF_FLUSH_LIST:
1700                count = buf_do_flush_list_batch(buf_pool, min_n, lsn_limit);
1701                break;
1702        default:
1703                ut_error;
1704        }
1705
1706        buf_pool_mutex_exit(buf_pool);
```

Flush the pages in the buffer pool

RW-lock in InnoDB

- The low-level object to represent and enforce **shared-access locks** to internal in-memory data structures

- RW-lock includes three types of locks
  - **S-locks** (shared locks)
  - **X-locks** (exclusive locks)
  - **SX-locks** (shared-exclusive locks)

	S	SX	X
S	Compatible	Compatible	Conflict
SX	Compatible	Conflict	Conflict
X	Conflict	Conflict	Conflict

RW-lock in InnoDB

- **S-lock (Shared-lock)**
  - provides **read access** to a common resource
- **X-lock (eXclusive-lock)**
  - provides **write access** to a common resource
  - while not permitting inconsistent reads by other threads
- **SX-lock (Shared-eXclusive lock)**
  - provides **write access** to a common resource
  - while permitting inconsistent **reads** by other threads
  - introduced in MySQL 5.7 to optimize concurrency and improve scalability for read-write workloads.

RW-lock in InnoDB

- Example code in InnoDB (S-lock)
  - buf/buf0buf.cc: buf_page_get_gen()

```
2776            rw_lock_s_lock(hash_lock);
 ...
2797            if (block == NULL) {
2798                    block = (buf_block_t*) buf_page_hash_get_low(
2799                            buf_pool, space, offset, fold);
2800            }
2801
2802            if (!block || buf_pool_watch_is_sentinel(buf_pool, &block->page)) {
2803                    rw_lock_s_unlock(hash_lock);
2804                    block = NULL;
2805            }
```

Search hash table

RW-lock in InnoDB

- Example code in InnoDB (X-lock)
  - buf/buf0buf.cc: buf_page_init_for_read()

```
3859                    rw_lock_x_lock(hash_lock);
 ...
3933                    HASH_INSERT(buf_page_t, hash, buf_pool->page_hash, fold,
3934                                bpage);
3935
3936                    rw_lock_x_unlock(hash_lock);
```

Insert a page into the hash table