DBMS Implementation : Buffer Manager & File Manager

woonhak.kang

woonagi319@skku.edu

Contents

- Buffer manager
  - Overview
  - Buffer Pool (buf0buf.cc)
  - Buffer Read (buf0read.cc)
  - LRU (buf0lru.cc)
  - Flusher (buf0flu.cc)
  - Doublewrite (buf0dblwr.cc)
- File manager
  - Overview
  - file system (fil0fil.cc)
  - OS specific impl. for file system (os0file.cc)

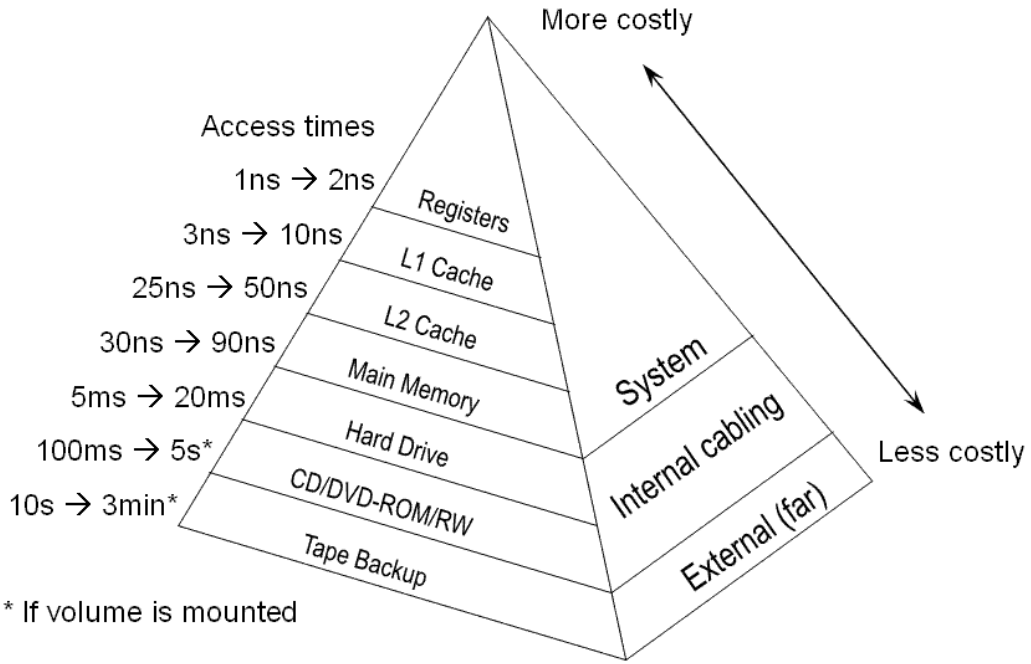
Changed Plan

- Plan
  - 1st week
    - Overview
    - Run DBMS and Tools
  - 2nd week
    - Code review for InnoDB initialize procedures (startup database)
  - 3rd week
    - Code review for InnoDB initialize procedures (startup database)
    - Buffer manager part1
  - 4th week
    - Buffer manager part2
    - File Storage manager
  - ~~5th week~~
    - ~~Transaction manager~~
    - ~~Log manager~~
- Term Project
  - Implement something in the MySQL
  - flash optimized techniques, NVM

BUFFER MANGER

Overview

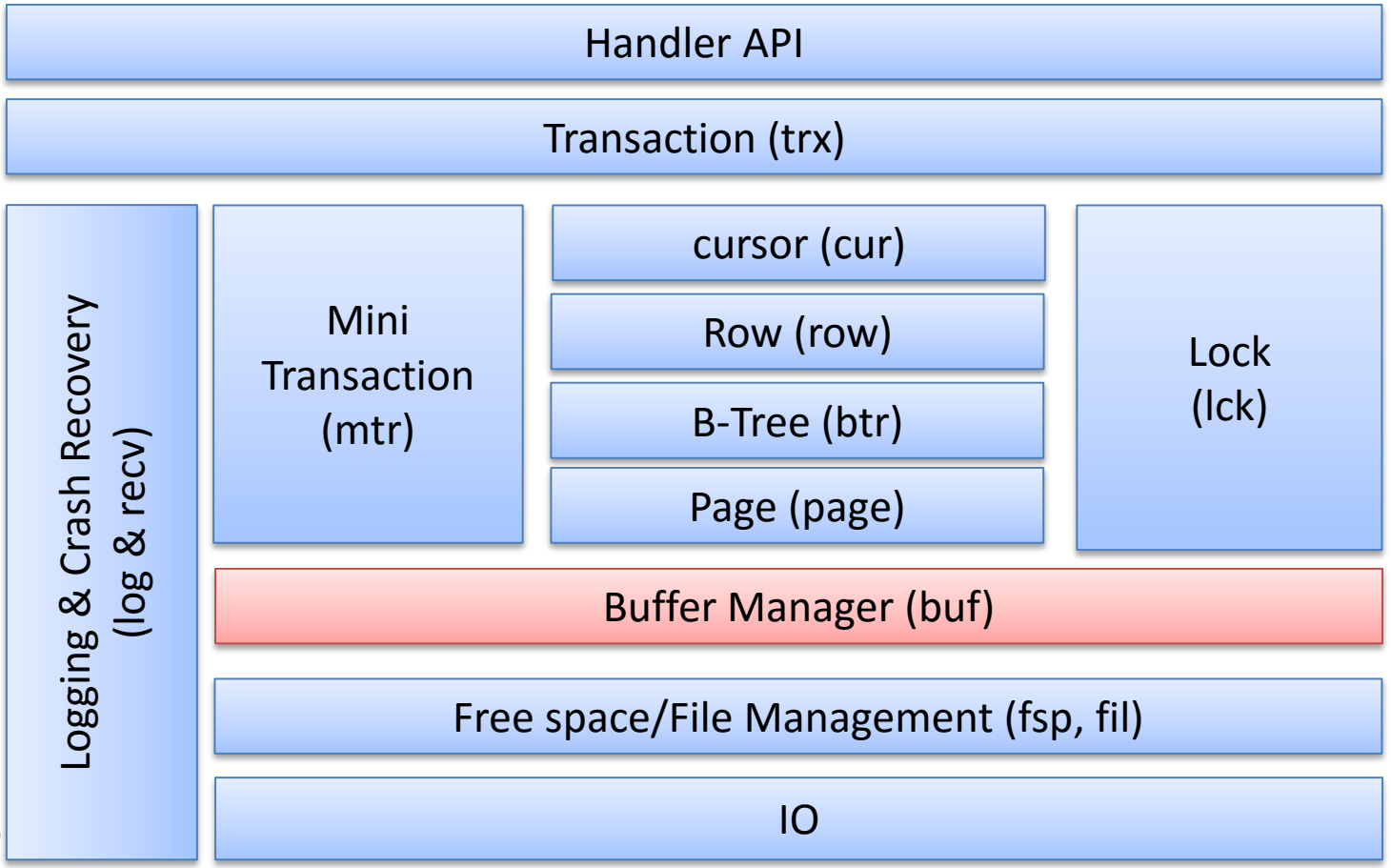
- Buffer pool
  - Considering memory hierarchy
  - Caching frequently accessed data into DRAM like a cache memory in CPU
  - Exploit locality



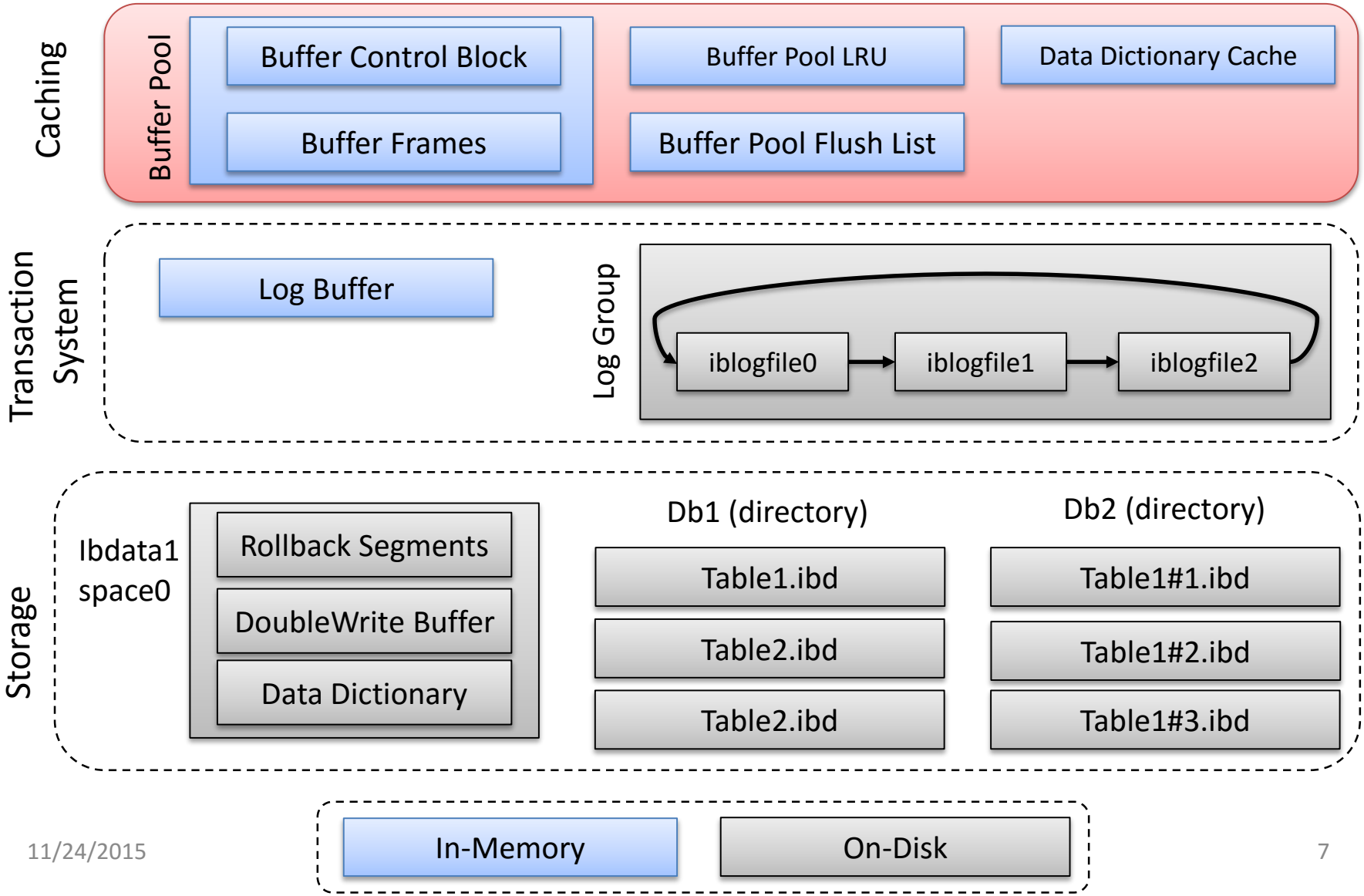
* If volume is mounted

Overview

- InnoDB Architecture - Buffer Manager

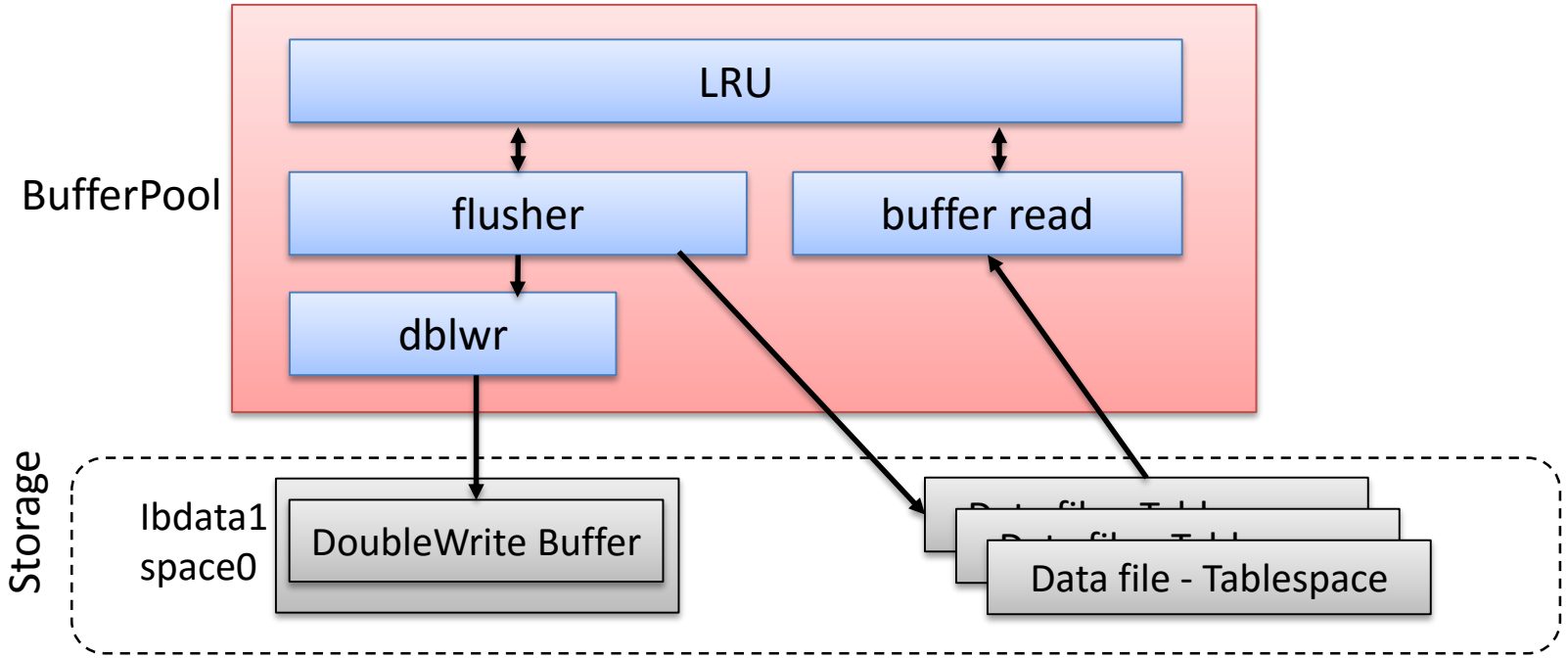


Overview


Caching
Buffer Pool
Buffer Control Block
Buffer Frames
Buffer Pool LRU
Buffer Pool Flush List
Data Dictionary Cache
Transaction System
Log Buffer
Log Group
iblogfile0
iblogfile1
iblogfile2
Storage
Ibdata1
space0
Rollback Segments
DoubleWrite Buffer
Data Dictionary
Db1 (directory)
Table1.ibd
Table2.ibd
Table2.ibd
Db2 (directory)
Table1#1.ibd
Table1#2.ibd
Table1#3.ibd
In-Memory
On-Disk

Overview

- Buffer Manager
  - Buffer Pool (buf0buf.cc) : Buffer Pool manager
  - Buffer Read (buf0read.cc) : Read Buffer
  - LRU (buf0lru.cc) : Buffer Replacement
  - Flusher (buf0flu.cc) : dirty page writer, background flusher
  - Doublewrite (buf0dblwr.cc) : Doublewrite



Overview

- UNIV_DEBUG symbol
  - if you are going to compile mysql with debug on
  - otherwise, UNIV_DEBUG is not defined
- Buffer pool
  - buffer_pool_t
  - buffer_block_t
- Buffer read
  - follow the read path of InnoDB
  - it has a couple of wrapper functions
  - list operations : Free, LRU
- Buffer flush
  - doublewrite
  - we will follow the single page flush path, or background flusher - LRU or flush list
  - list operations : Free, Flush, LRU

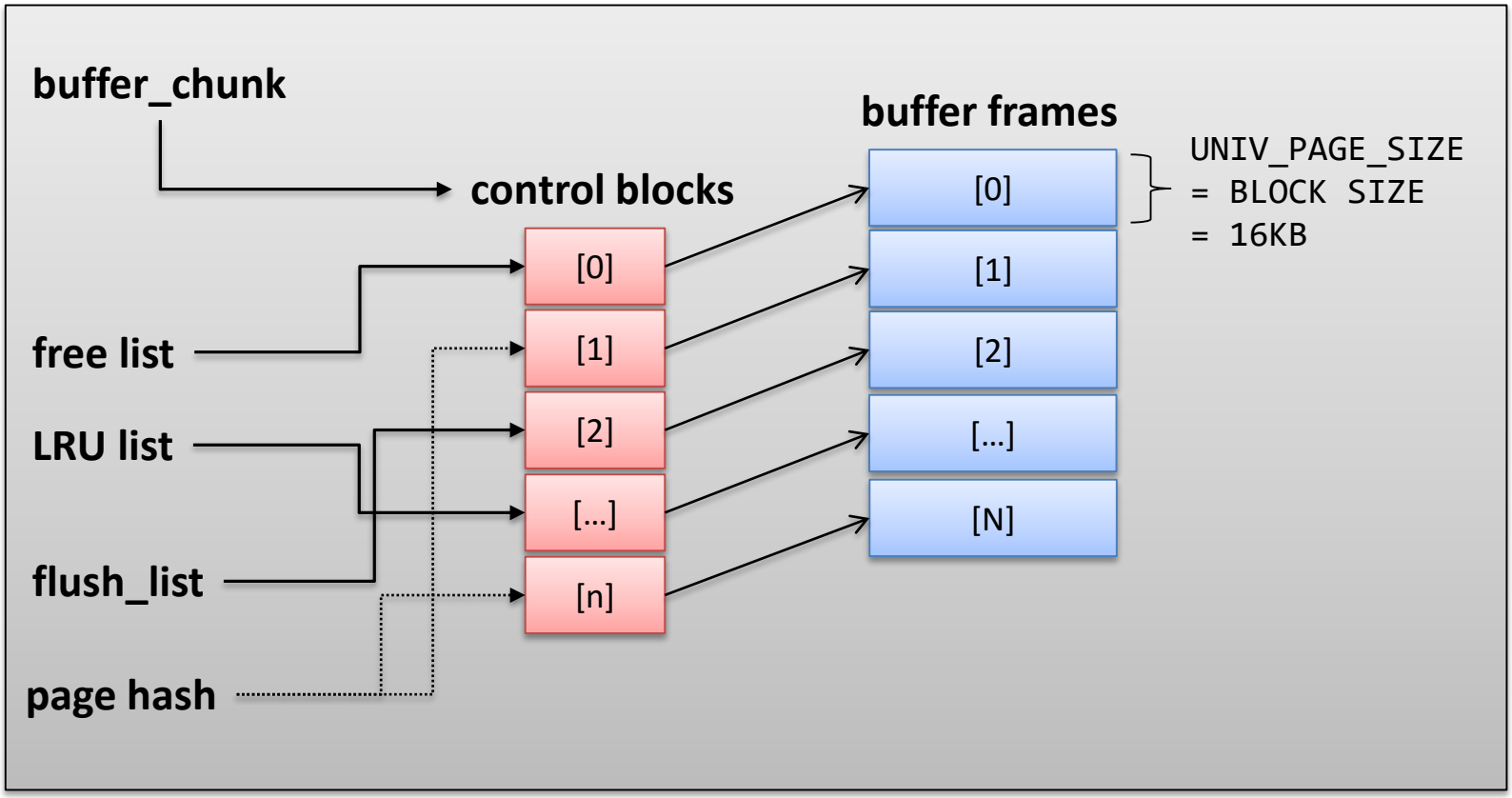
BUFFER POOL

Buffer Pool Struct

- The buffer buf_pool contains a single mutex
  - buf_pool->mutex : protects all the control data structures of the buf_pool
- The content of a buffer frame is protected by a separate read-write lock in its control block
  - buf_block->mutex : protects specific buffer block (frame)
  - These locks can be locked and unlocked without owning the buf_pool->mutex.
- The buf_pool->mutex is a hot-spot in main memory
  - causing a lot of memory bus traffic on multiprocessor systems when processors alternately access the mutex
  - A solution to reduce mutex contention
    - to create a separate mutex for the page hash table
    - hash_table_t->mutex

Buffer Pool

- Control block and buffer frame



Buffer Pool Struct

- include/buf0buf.h:1823, buf_pool_t

```
1822
1823 struct buf_pool_t{
1824
1825   /** @name General fields */
1826   /* @{ */
1827   ib_mutex_t  mutex;    /*!< Buffer pool mutex of this
1828           instance */
1829   ib_mutex_t  zip_mutex;  /*!< Zip mutex of this
1830           pool instance, protects compressed
1831           only pages (of type buf_page_t, not
1832           buf_block_t */
1833   ulint   instance_no;  /*!< Array index of this buffer
1834           pool instance */
1835   ulint   old_pool_size;  /*!< Old pool size in bytes */
1836   ulint   curr_pool_size; /*!< Current pool size in bytes */
1837   ulint   LRU_old_ratio;  /*!< Reserve this much of the buffer
1838           pool for "old" blocks */
1839 #ifdef UNIV_DEBUG
1840   ulint   buddy_n_frames; /*!< Number of frames allocated from
1841           the buffer pool to the buddy system */
1842 #endif
1843 #if defined UNIV_DEBUG || defined UNIV_BUF_DEBUG
1844   ulint   mutex_exit_forbidden; /*!< Forbid release mutex */
1845 #endif
```

buffer pool mutex

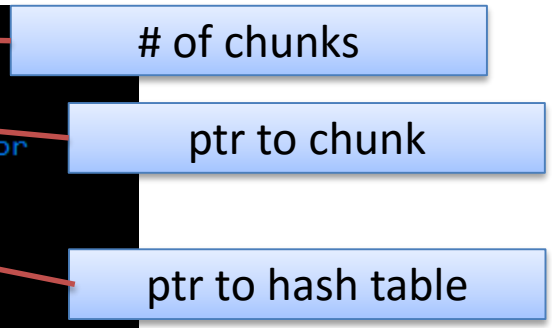
instance number

size of buffer pool

Buffer Pool Struct

- include/buf0buf.h:1823, buf_pool_t

```
1846    ulint   n_chunks; /*!< number of buffer pool chunks */
1847    buf_chunk_t*  chunks;   /*!< buffer pool chunks */
1848    ulint   curr_size;  /*!< current pool size in pages */
1849    hash_table_t* page_hash;  /*!< hash table of buf_page_t or
1850            buf_block_t file pages,
1851            buf_page_in_file() == TRUE,
1852            indexed by (space_id, offset).
1853            page_hash is protected by an
1854            array of mutexes.
1855            Changes in page_hash are protected
1856            by buf_pool->mutex and the relevant
1857            page_hash mutex. Lookups can happen
1858            while holding the buf_pool->mutex or
1859            the relevant page_hash mutex. */
1860    hash_table_t* zip_hash; /*!< hash table of buf_block_t blocks
1861            whose frames are allocated to the
1862            zip buddy system,
1863            indexed by block->frame */
1864    ulint   n_pend_reads; /*!< number of pending read
1865            operations */
1866    ulint   n_pend_unzip; /*!< number of pending decompressions */
1867
1868    time_t    last_printout_time;
1869            /*!< when buf_print_io was last time
1870            called */
```



Buffer Pool Struct

- include/buf0buf.h:1823, buf_pool_t

```
1871   buf_buddy_stat_t buddy_stat[BUF_BUDDY_SIZES_MAX + 1];
1872           /*!< Statistics of buddy system,
1873           indexed by block size */
1874   buf_pool_stat_t stat;   /*!< current statistics */
1875   buf_pool_stat_t old_stat; /*!< old statistics */
1876
```

Buffer pool stat (hit ratio etc.)

Lists of buffer blocks

- Free list
  - buf_pool->free
- LRU list
  - contains all the blocks holding a file page except those for which the bufferfix count is non-zero
  - The pages are in the LRU list roughly in the order of the last access to the page, so that the oldest pages are at the end of the list
- Flush list
  - contains the blocks holding file pages that have been modified in the memory but not written to disk yet
  - The block with the oldest modification which has not yet been written to disk is at the end of the chain
  - The access to this list is protected by buf_pool->flush_list_mutex

Buffer Pool Struct

- include/buf0buf.h:1823, buf_pool_t

```
1879   /** @name Page flushing algorithm fields */
1880
1881   /* @{ */
1882
1883   ib_mutex_t  flush_list_mutex;/*!< mutex protecting the
1884           flush list access. This mutex
1885           protects flush_list, flush_rbt
1886           and bpage::list pointers when
1887           the bpage is on flush_list. It
1888           also protects writes to
1889           bpage::oldest_modification and
1890           flush_list_hp */
1891   const buf_page_t* flush_list_hp;/*!< "hazard pointer"
1892           used during scan of flush_list
1893           while doing flush list batch.
1894           Protected by flush_list_mutex */
1895   UT_LIST_BASE_NODE_T(buf_page_t) flush_list;
1896           /*!< base node of the modified block
1897           list */
1898   ibool   init_flush[BUF_FLUSH_N_TYPES];
1899           /*!< this is TRUE when a flush of the
1900           given type is being initialized */
1901   ulint   n_flush[BUF_FLUSH_N_TYPES];
1902           /*!< this is the number of pending
1903           writes in the given flush type */
1904   os_event_t  no_flush[BUF_FLUSH_N_TYPES];
1905           /*!< this is in the set state
1906           when there is no flush batch
1907           of the given type running */
```

flush list mutex

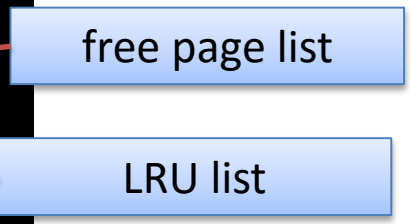
flush list

status for each flush types

Buffer Pool Struct

- include/buf0buf.h:1823, buf_pool_t

```
1943   /** @name LRU replacement algorithm fields */
1944   /* @{ */
1945
1946   UT_LIST_BASE_NODE_T(buf_page_t) free;
1947           /*!< base node of the free
1948           block list */
1949   UT_LIST_BASE_NODE_T(buf_page_t) LRU;
1950           /*!< base node of the LRU list */
1951   buf_page_t* LRU_old;  /*!< pointer to the about
1952           LRU_old_ratio/BUF_LRU_OLD_RATIO_DIV
1953           oldest blocks in the LRU list;
1954           NULL if LRU length less than
1955           BUF_LRU_OLD_MIN_LEN;
1956           NOTE: when LRU_old != NULL, its length
1957           should always equal LRU_old_len */
1958   ulint   LRU_old_len;  /*!< length of the LRU list from
1959           the block to which LRU_old points
1960           onward, including that block;
1961           see buf0lru.cc for the restrictions
1962           on this value; 0 if LRU_old == NULL;
1963           NOTE: LRU_old_len must be adjusted
1964           whenever LRU_old shrinks or grows! */
1965
1966   UT_LIST_BASE_NODE_T(buf_block_t) unzip_LRU;
1967           /*!< base node of the
1968           unzip_LRU list */
1969
```



Buffer Pool Struct

- include/buf0buf.h:1823, buf_pool_t

```
1970   /* @} */
1971   /** @name Buddy allocator fields
1972   The buddy allocator is used for allocating compressed page
1973   frames and buf_page_t descriptors of blocks that exist
1974   in the buffer pool only in compressed form. */
1975   /* @{ */
1976 #if defined UNIV_DEBUG || defined UNIV_BUF_DEBUG
1977   UT_LIST_BASE_NODE_T(buf_page_t) zip_clean;
1978           /*!< unmodified compressed pages */
1979 #endif /* UNIV_DEBUG || UNIV_BUF_DEBUG */
1980   UT_LIST_BASE_NODE_T(buf_buddy_free_t) zip_free[BUF_BUDDY_SIZES_MAX];
1981           /*!< buddy free lists */
1982
1983   buf_page_t*     watch;
1984           /*!< Sentinel records for buffer
1985           pool watches. Protected by
1986           buf_pool->mutex. */
1987
1988 #if BUF_BUDDY_LOW > UNIV_ZIP_SIZE_MIN
1989 # error "BUF_BUDDY_LOW > UNIV_ZIP_SIZE_MIN"
1990 #endif
1991   /* @} */
1992 };
```

Sentinel records for buffer pool watches

Buffer pool init

- buf/buf0buf.cc:1384, buf_pool_init()

```
1384 buf_pool_init(
1385 /*===========*/
1386   ulint total_size, /*!< in: size of the total pool in bytes */
1387   ulint n_instances)  /*!< in: number of instances */
1388 {
1389   ulint   i;
1390   const ulint size  = total_size / n_instances;
1391
1392   ut_ad(n_instances > 0);
1393   ut_ad(n_instances <= MAX_BUFFER_POOLS);
1394   ut_ad(n_instances == srv_buf_pool_instances);
1395
1396   buf_pool_ptr = (buf_pool_t*) mem_zalloc(
1397     n_instances * sizeof *buf_pool_ptr);
1398
1399   for (i = 0; i < n_instances; i++) {
1400     buf_pool_t* ptr = &buf_pool_ptr[i];
1401
1402     if (buf_pool_init_instance(ptr, size, i) != DB_SUCCESS) {
1403
1404       /* Free all the instances created so far. */
1405       buf_pool_free(i);
1406
1407       return(DB_ERROR);
1408     }
1409   }
1410
1411   buf_pool_set_sizes();
1412   buf_LRU_old_ratio_update(100 * 3/ 8, FALSE);
1413
1414   btr_search_sys_create(buf_pool_get_curr_size() / sizeof(void*) / 64);
1415
1416   return(DB_SUCCESS);
1417 }
```

buffer pool init per instance

Buffer pool init

- buf/buf0buf.cc:1253, buf_pool_instance()

```
1248 /********************************************************************//**
1249 Initialize a buffer pool instance.
1250 @return DB_SUCCESS if all goes well. */
1251 UNIV_INTERN
1252 ulint
1253 buf_pool_init_instance(
1254 /*===================*/
1255   buf_pool_t* buf_pool, /*!< in: buffer pool instance */
1256   ulint   buf_pool_size,  /*!< in: size in bytes */
1257   ulint   instance_no)  /*!< in: id of the instance */
1258 {
1259   ulint   i;
1260   buf_chunk_t*  chunk;
1261
1262   /* 1. Initialize general fields
1263   ------------------------------- */
1264   mutex_create(buf_pool_mutex_key,
1265          &buf_pool->mutex, SYNC_BUF_POOL);
1266   mutex_create(buf_pool_zip_mutex_key,
1267          &buf_pool->zip_mutex, SYNC_BUF_BLOCK);
1268
1269   buf_pool_mutex_enter(buf_pool);
1270
1271   if (buf_pool_size > 0) {
1272     buf_pool->n_chunks = 1;
1273
1274     buf_pool->chunks = chunk =
1275       (buf_chunk_t*) mem_zalloc(sizeof *chunk);
1276
1277     UT_LIST_INIT(buf_pool->free);
```

create mutex and enter

Buffer pool init

- buf/buf0buf.cc:1253, buf_pool_instance()

```
1277    UT_LIST_INIT(buf_pool->free);
1278
1279    if (!buf_chunk_init(buf_pool, chunk, buf_pool_size)) {
1280      mem_free(chunk);
1281      mem_free(buf_pool);
1282
1283      buf_pool_mutex_exit(buf_pool);
1284
1285      return(DB_ERROR);
1286    }
1287
1288    buf_pool->instance_no = instance_no;
1289    buf_pool->old_pool_size = buf_pool_size;
1290    buf_pool->curr_size = chunk->size;
1291    buf_pool->curr_pool_size = buf_pool->curr_size * UNIV_PAGE_SIZE;
```

init buffer chunk - see this later

Buffer pool init

- buf/buf0buf.cc:1253, buf_pool_instance()

```
1293    /* Number of locks protecting page_hash must be a
1294    power of two */
1295    srv_n_page_hash_locks = static_cast<ulong>(
1296         ut_2_power_up(srv_n_page_hash_locks));
1297    ut_a(srv_n_page_hash_locks != 0);
1298    ut_a(srv_n_page_hash_locks <= MAX_PAGE_HASH_LOCKS);
1299
1300    buf_pool->page_hash = ha_create(2 * buf_pool->curr_size,
1301            srv_n_page_hash_locks,
1302            MEM_HEAP_FOR_PAGE_HASH,
1303            SYNC_BUF_PAGE_HASH);
1304
1305    buf_pool->zip_hash = hash_create(2 * buf_pool->curr_size);
1306
1307    buf_pool->last_printout_time = ut_time();
```

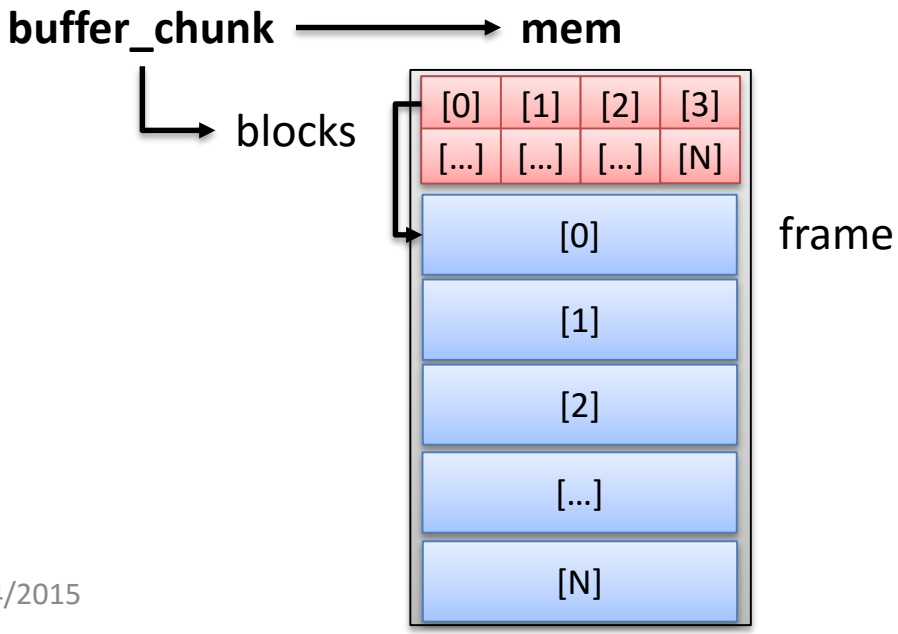
create page hash table

srv_n_page_hash_locks = 16,
so page hash table divided into 16

Buffer pool init

- buffer chunk

```
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:1041, chunk_init()

```
1036 /********************************************************************//**
1037 Allocates a chunk of buffer frames.
1038 @return chunk, or NULL on failure */
1039 static
1040 buf_chunk_t*
1041 buf_chunk_init(
1042 /*===========*/
1043   buf_pool_t* buf_pool, /*!< in: buffer pool instance */
1044   buf_chunk_t*  chunk,    /*!< out: chunk of buffers */
1045   ulint   mem_size) /*!< in: requested size in bytes */
1046 {
1047   buf_block_t*  block;
1048   byte*   frame;
1049   ulint   i;
1050
1051   /* Round down to a multiple of page size,
1052   although it already should be. */
1053   mem_size = ut_2pow_round(mem_size, UNIV_PAGE_SIZE);
1054   /* Reserve space for the block descriptors. */
1055   mem_size += ut_2pow_round((mem_size / UNIV_PAGE_SIZE) * (sizeof *block)
1056           + (UNIV_PAGE_SIZE - 1), UNIV_PAGE_SIZE);
1057
1058   chunk->mem_size = mem_size;
1059   chunk->mem = os_mem_alloc_large(&chunk->mem_size);
1060
1061   if (UNIV_UNLIKELY(chunk->mem == NULL)) {
1062
1063     return(NULL);
1064   }
```

allocate chunk mem (blocks + frames)

Buffer pool init

- buf/buf0buf.cc:1041, chunk_init()

```
1066  /* Allocate the block descriptors from
1067  the start of the memory block. */
1068  chunk->blocks = (buf_block_t*) chunk->mem;
1069
1070  /* Align a pointer to the first frame.  Note that when
1071  os_large_page_size is smaller than UNIV_PAGE_SIZE,
1072  we may allocate one fewer block than requested.  When
1073  it is bigger, we may allocate more blocks than requested. */
1074
1075  frame = (byte*) ut_align(chunk->mem, UNIV_PAGE_SIZE);
1076  chunk->size = chunk->mem_size / UNIV_PAGE_SIZE
1077    - (frame != chunk->mem);
1078
1079  /* Subtract the space needed for block descriptors. */
1080  {
1081    ulint size = chunk->size;
1082
1083    while (frame < (byte*) (chunk->blocks + size)) {
1084      frame += UNIV_PAGE_SIZE;
1085      size--;
1086    }
1087
1088    chunk->size = size;
1089  }
```

allocate control blocks

allocate frame
(page size aligned)

Buffer pool init

- `buf/buf0buf.cc:1041, chunk_init()`

```
1091   /* Init block structs and assign frames for them. Then we
1092   assign the frames to the first blocks (we already mapped the
1093   memory above). */
1094
1095   block = chunk->blocks;
1096
1097   for (i = chunk->size; i--; ) {
1098
1099     buf_block_init(buf_pool, block, frame);
1100     UNIV_MEM_INVALID(block->frame, UNIV_PAGE_SIZE);
1101
1102     /* Add the block to the free list */
1103     UT_LIST_ADD_LAST(list, buf_pool->free, (&block->page));
1104
1105     ut_d(block->page.in_free_list = TRUE);
1106     ut_ad(buf_pool_from_block(block) == buf_pool);
1107
1108     block++;
1109     frame += UNIV_PAGE_SIZE;
1110   }
1111
1112 #ifdef PFS_GROUP_BUFFER_SYNC
1113   pfs_register_buffer_block(chunk);
1114 #endif
1115   return(chunk);
```

init control block

add all blocks to free list

Buffer pool init

- buf/buf0buf.cc:1384, buf_pool_init()

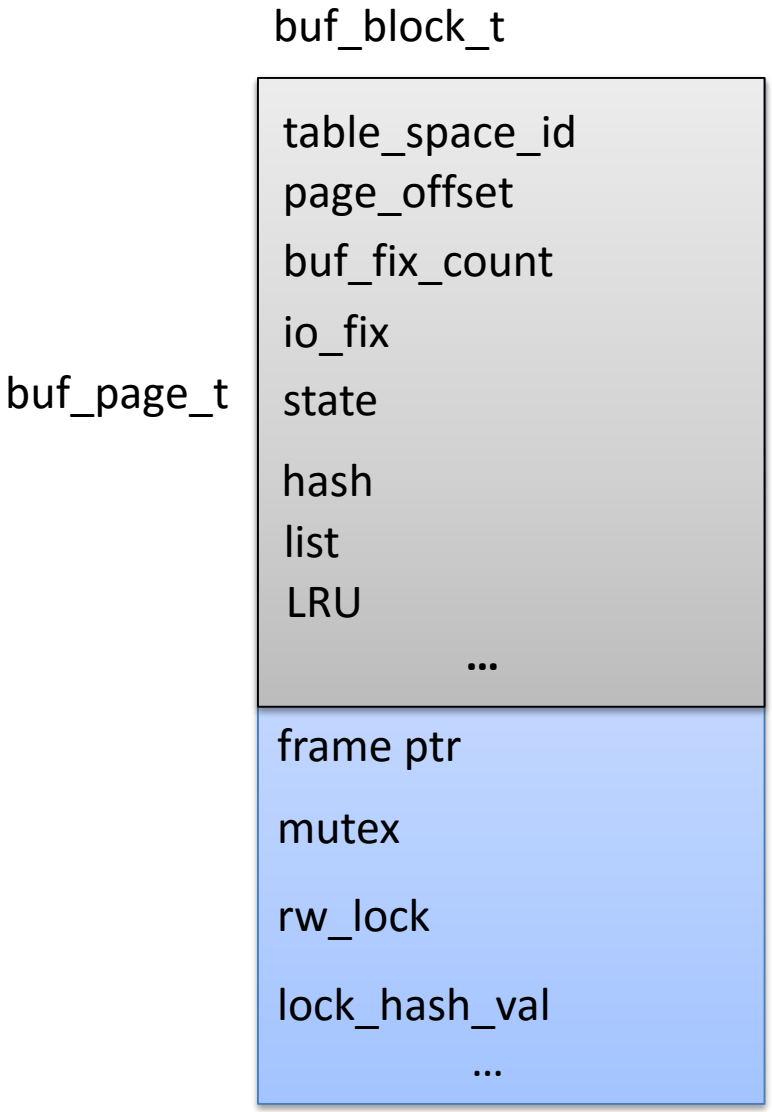
```
1384 buf_pool_init(
1385 /*===========*/
1386   ulint total_size, /*!< in: size of the total pool in bytes */
1387   ulint n_instances)  /*!< in: number of instances */
1388 {
1389   ulint   i;
1390   const ulint size  = total_size / n_instances;
1391
1392   ut_ad(n_instances > 0);
1393   ut_ad(n_instances <= MAX_BUFFER_POOLS);
1394   ut_ad(n_instances == srv_buf_pool_instances);
1395
1396   buf_pool_ptr = (buf_pool_t*) mem_zalloc(
1397     n_instances * sizeof *buf_pool_ptr);
1398
1399   for (i = 0; i < n_instances; i++) {
1400     buf_pool_t* ptr = &buf_pool_ptr[i];
1401
1402     if (buf_pool_init_instance(ptr, size, i) != DB_SUCCESS) {
1403
1404       /* Free all the instances created so far. */
1405       buf_pool_free(i);
1406
1407       return(DB_ERROR);
1408     }
1409   }
1410
1411   buf_pool_set_sizes();
1412   buf_LRU_old_ratio_update(100 * 3/ 8, FALSE);
1413
1414   btr_search_sys_create(buf_pool_get_curr_size() / sizeof(void*) / 64);
1415
1416   return(DB_SUCCESS);
1417 }
```

set old ratio : 3/8 -> for mid-point insertion

Buffer Control Block (BCB)

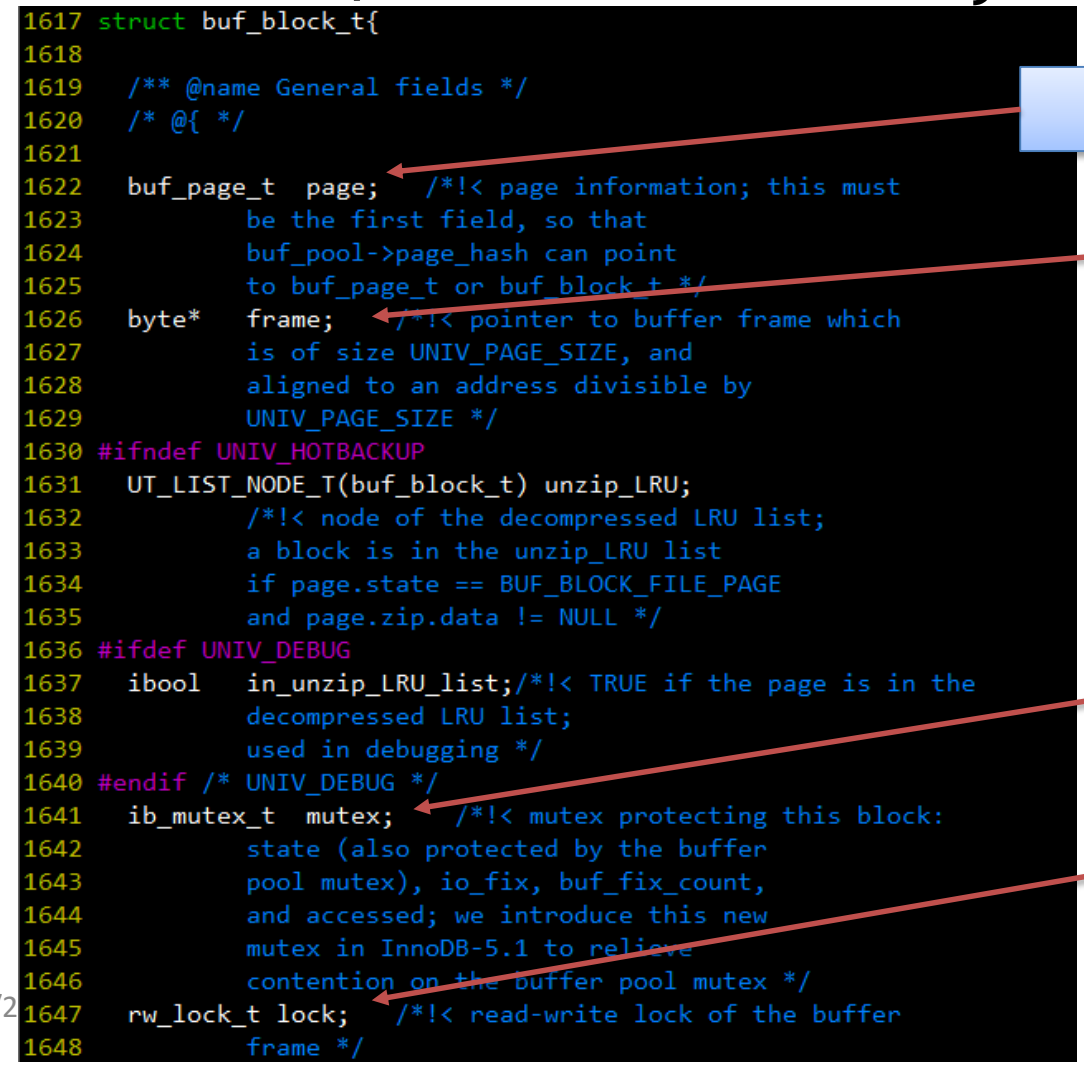
- The control block contains
  - Read-write lock
  - Buffer fix count
    - which is incremented when a thread wants a file page to be fixed in a buffer frame.
    - The bufferfix operation does not lock the contents of the frame
  - Page frame
  - File pages
    - put to a hash table according to the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

Buffer Control Block (BCB)



BCB struct

- include/buf0buf.h:1617, buf_block_t

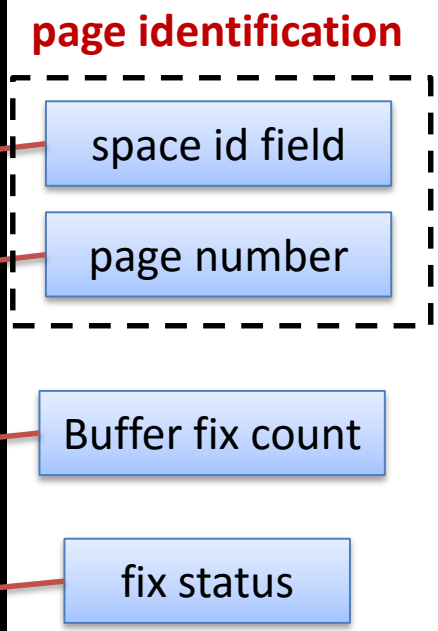


```
1617 struct buf_block_t{
1618
1619   /** @name General fields */
1620   /* @{ */
1621
1622   buf_page_t  page;    /*!< page information; this must
1623           be the first field, so that
1624           buf_pool->page_hash can point
1625           to buf_page_t or buf_block_t */
1626   byte*    frame;    /*!< pointer to buffer frame which
1627           is of size UNIV_PAGE_SIZE, and
1628           aligned to an address divisible by
1629           UNIV_PAGE_SIZE */
1630 #ifndef UNIV_HOTBACKUP
1631   UT_LIST_NODE_T(buf_block_t) unzip_LRU;
1632           /*!< node of the decompressed LRU list;
1633           a block is in the unzip_LRU list
1634           if page.state == BUF_BLOCK_FILE_PAGE
1635           and page.zip.data != NULL */
1636 #ifdef UNIV_DEBUG
1637   ibool    in_unzip_LRU_list;/*!< TRUE if the page is in the
1638           decompressed LRU list;
1639           used in debugging */
1640 #endif /* UNIV_DEBUG */
1641   ib_mutex_t  mutex;     /*!< mutex protecting this block:
1642           state (also protected by the buffer
1643           pool mutex), io_fix, buf_fix_count,
1644           and accessed; we introduce this new
1645           mutex in InnoDB-5.1 to relieve
1646           contention on the buffer pool mutex */
1647   rw_lock_t lock;    /*!< read-write lock of the buffer
1648           frame */
```

Buffer Page Struct

- include/buf0buf.h:1443, buf_page_t

```
1443 struct buf_page_t{
1444   /** @name General fields
1445   None of these bit-fields must be modified without holding
1446   buf_page_get_mutex() [buf_block_t::mutex or
1447   buf_pool->zip_mutex], since they can be stored in the same
1448   machine word.  Some of these fields are additionally protected
1449   by buf_pool->mutex. */
1450   /* @{ */
1451
1452   ib_uint32_t space;    /*!< tablespace id; also protected
1453           by buf_pool->mutex. */
1454   ib_uint32_t offset;   /*!< page number; also protected
1455           by buf_pool->mutex. */
1456   /** count of how manyfold this block is currently bufferfixed */
1457 #ifdef PAGE_ATOMIC_REF_COUNT
1458   ib_uint32_t buf_fix_count;
1459
1460   /** type of pending I/O operation; also protected by
1461   buf_pool->mutex for writes only @see enum buf_io_fix */
1462   byte    io_fix;
1463
1464   byte    state;
1465 #else
1466   unsigned  buf_fix_count:19;
```



Buffer Page Struct

- include/buf0buf.h:1443, buf_page_t

```
1483 #ifndef UNIV_HOTBACKUP
1484   unsigned  flush_type:2; /*!< if this block is currently being
1485           flushed to disk, this tells the
1486           flush_type.
1487           @see buf_flush_t */
1488   unsigned  buf_pool_index:6;/*!< index number of the buffer pool
1489           that this block belongs to */
1490 # if MAX_BUFFER_POOLS > 64
1491 #  error "MAX_BUFFER_POOLS > 64; redefine buf_pool_in
1492 # endif
1493   /* @} */
1494 #endif /* !UNIV_HOTBACKUP */
1495   page_zip_des_t  zip;    /*!< compressed page; zip.data
1496           (but not the data it points to) is
1497           also protected by buf_pool->mutex;
1498           state == BUF_BLOCK_ZIP_PAGE and
1499           zip.data == NULL means an active
1500           buf_pool->watch */
1501 #ifndef UNIV_HOTBACKUP
1502   buf_page_t* hash;   /*!< node used in chaining to
1503           buf_pool->page_hash or
1504           buf_pool->zip_hash */
```

flush type

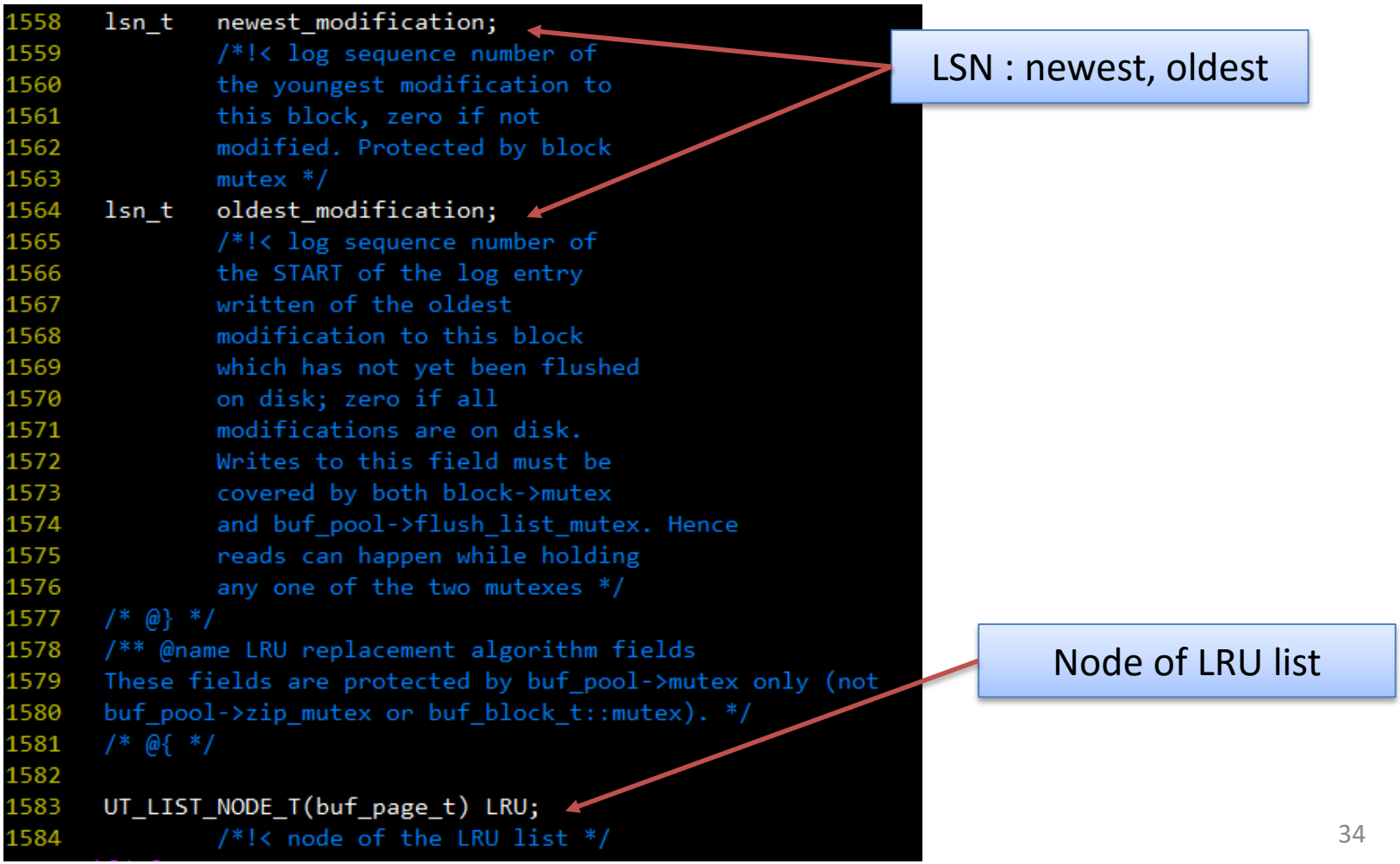
buf pool index - kind of backward reference

node of page hash table

Buffer Page Struct

- include/buf0buf.h:1443, buf_page_t

```
1558    lsn_t    newest_modification;
1559             /*!< log sequence number of
1560             the youngest modification to
1561             this block, zero if not
1562             modified. Protected by block
1563             mutex */
1564    lsn_t    oldest_modification;
1565             /*!< log sequence number of
1566             the START of the log entry
1567             written of the oldest
1568             modification to this block
1569             which has not yet been flushed
1570             on disk; zero if all
1571             modifications are on disk.
1572             Writes to this field must be
1573             covered by both block->mutex
1574             and buf_pool->flush_list_mutex. Hence
1575             reads can happen while holding
1576             any one of the two mutexes */
1577    /* @} */
1578    /** @name LRU replacement algorithm fields
1579    These fields are protected by buf_pool->mutex only (not
1580    buf_pool->zip_mutex or buf_block_t::mutex). */
1581    /* @{ */
1582
1583    UT_LIST_NODE_T(buf_page_t) LRU;
1584             /*!< node of the LRU list */
```



Buffer Page Struct

- include/buf0buf.h:1443, buf_page_t

```
1590    unsigned  old:1;    /*!< TRUE if the block is in the old
1591            blocks in buf_pool->LRU_old */
1592    unsigned  freed_page_clock:31;/*!< the value of
1593            buf_pool->freed_page_clock
1594            when this block was the last
1595            time put to the head of the
1596            LRU list; a thread is allowed
1597            to read this for heuristic
1598            purposes without holding any
1599            mutex or latch */
1600    /* @} */
1601    unsigned  access_time;  /*!< time of first access, or
1602            0 if the block was never accessed
1603            in the buffer pool. Protected by
1604            block mutex */
```

Information used in replacement

Buffer Control block init

- buf/buf0buf.cc:971, buf_block_init()

```
967 /********************************************************************//**
968 Initializes a buffer control block when the buf_pool is created. */
969 static
970 void
971 buf_block_init(
972 /*===========*/
973   buf_pool_t* buf_pool, /*!< in: buffer pool instance */
974   buf_block_t*  block,    /*!< in: pointer to control block */
975   byte*   frame)    /*!< in: pointer to buffer frame */
976 {
977   UNIV_MEM_DESC(frame, UNIV_PAGE_SIZE);
978
979   block->frame = frame;
980
981   block->page.buf_pool_index = buf_pool_index(buf_pool);
982   block->page.state = BUF_BLOCK_NOT_USED;
983   block->page.buf_fix_count = 0;
984   block->page.io_fix = BUF_IO_NONE;
985
986   block->modify_clock = 0;
```

set data frame

page.buf_pool_index : back pointer
page.state : current status
page.buf_fix_count : reference count
page.io_fix : block fix (Shared/eXclusive)

Buffer Control block init

- buf/buf0buf.cc:971, buf_block_init()

```
1008 #if defined PFS_SKIP_BUFFER_MUTEX_RWLOCK || defined PFS_GROUP_BUFFER_SYNC
1009   /* If PFS_SKIP_BUFFER_MUTEX_RWLOCK is defined, skip registration
1010   of buffer block mutex/rwlock with performance schema. If
1011   PFS_GROUP_BUFFER_SYNC is defined, skip the registration
1012   since buffer block mutex/rwlock will be registered later in
1013   pfs_register_buffer_block() */
1014
1015   mutex_create(PFS_NOT_INSTRUMENTED, &block->mutex, SYNC_BUF_BLOCK);
1016   rw_lock_create(PFS_NOT_INSTRUMENTED, &block->lock, SYNC_LEVEL_VARYING);
1017
1018 # ifdef UNIV_SYNC_DEBUG
1019   rw_lock_create(PFS_NOT_INSTRUMENTED,
1020            &block->debug_latch, SYNC_NO_ORDER_CHECK);
1021 # endif /* UNIV_SYNC_DEBUG */
1022
1023 #else /* PFS_SKIP_BUFFER_MUTEX_RWLOCK || PFS_GROUP_BUFFER_SYNC */
1024   mutex_create(buffer_block_mutex_key, &block->mutex, SYNC_BUF_BLOCK);
1025   rw_lock_create(buf_block_lock_key, &block->lock, SYNC_LEVEL_VARYING);
1026
1027 # ifdef UNIV_SYNC_DEBUG
1028   rw_lock_create(buf_block_debug_latch_key,
1029            &block->debug_latch, SYNC_NO_ORDER_CHECK);
1030 # endif /* UNIV_SYNC_DEBUG */
1031 #endif /* PFS_SKIP_BUFFER_MUTEX_RWLOCK || PFS_GROUP_BUFFER_SYNC */
1032
1033   ut_ad(rw_lock_validate(&(block->lock)));
1034 }
```

create block mutex and rw_lock

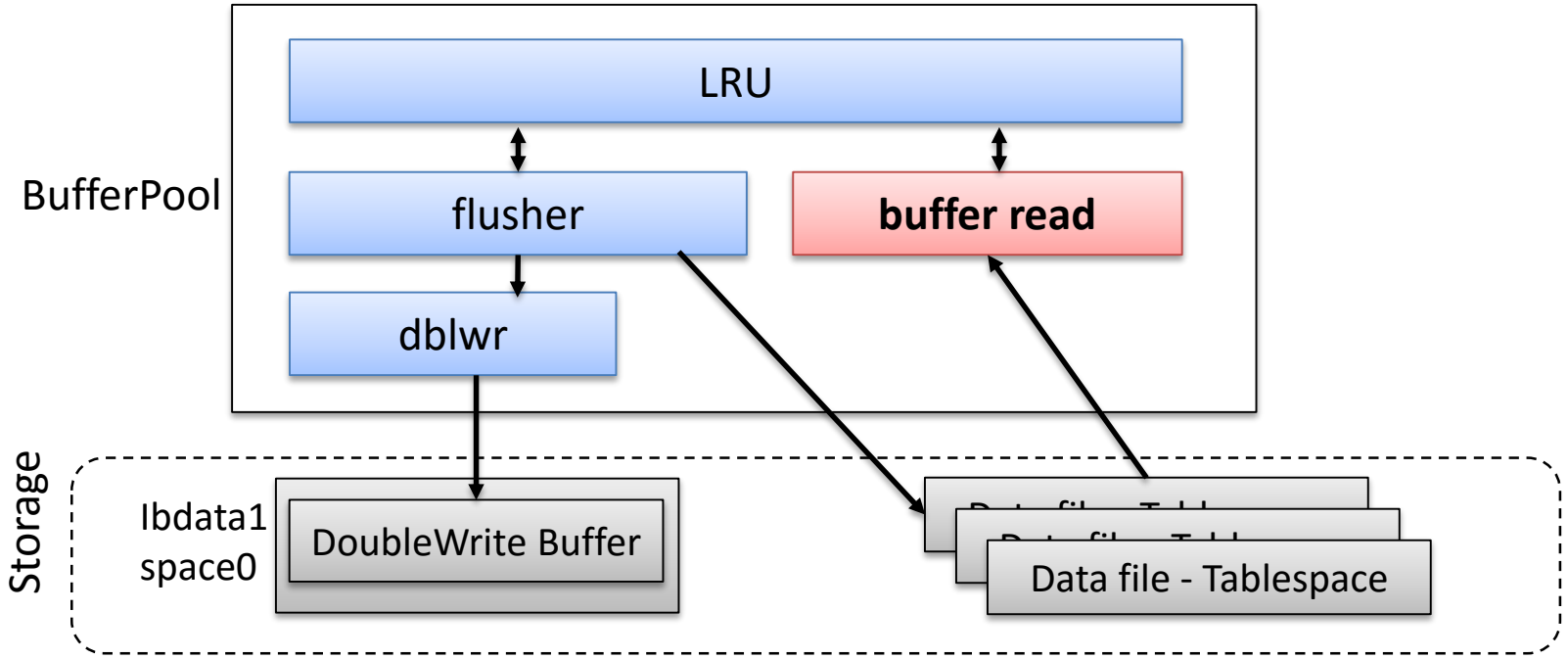
BUFFER READ - READ A PAGE

Buffer Read

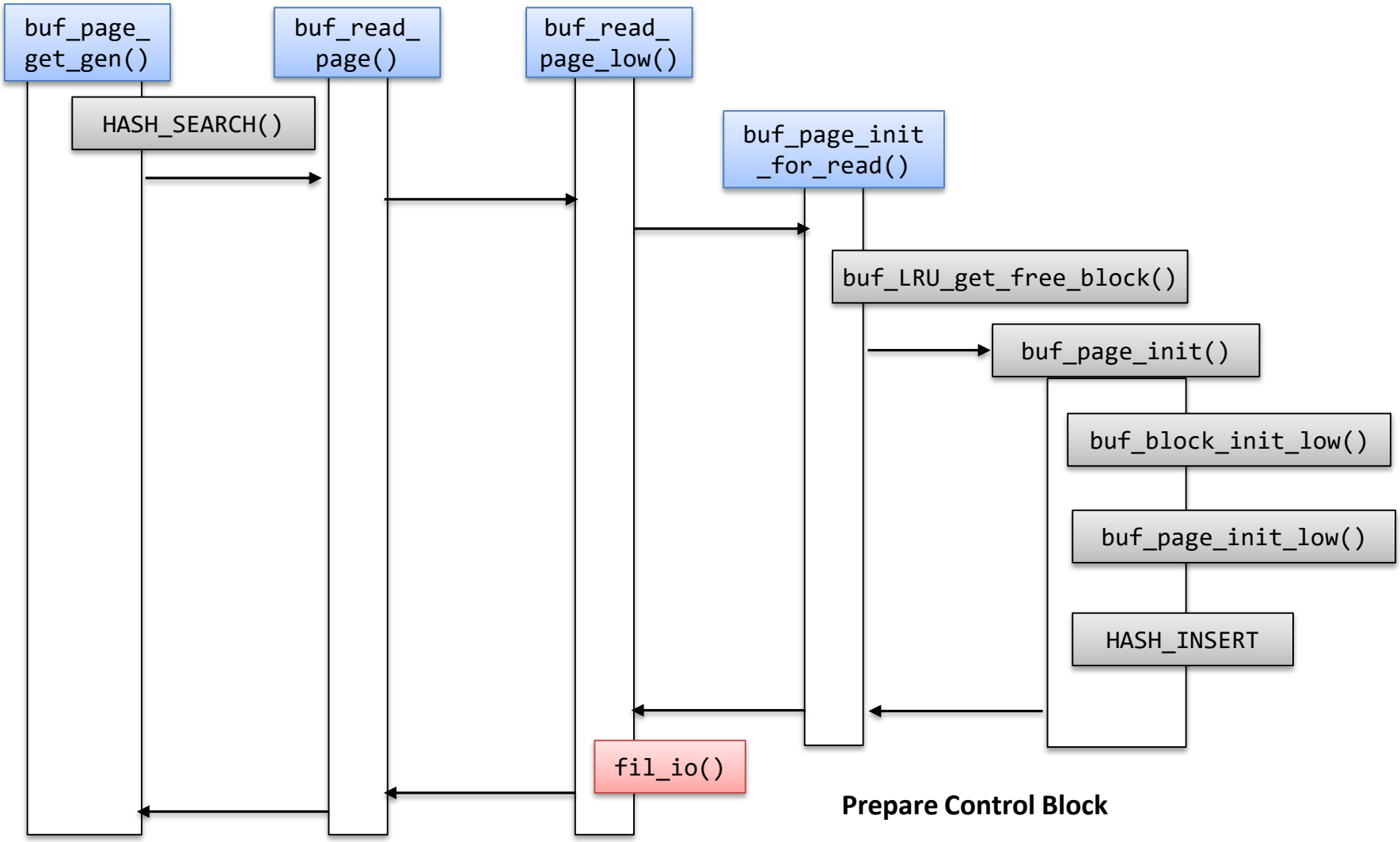
- We will focus on only buffer manager part
  - so will skip data block read through file system manager in MySQL
- Read a page (buf0rea.cc)
  - Find a certain page in the buffer pool using hash table
  - if it is not in the buffer pool, then read a block from the storage
    - Allocate a free block for read (include buffer block)
    - Two cases
      - buffer pool has free blocks
      - don’t have a free block
    - Read a page

Overview

- Buffer Manager
  - Buffer Pool (buf0buf.cc) : Buffer Pool manager
  - **Buffer Read (buf0read.cc) : Read Buffer**
  - LRU (buf0lru.cc) : Buffer Replacement
  - Flusher (buf0flu.cc) : dirty page writer, background flusher
  - Doublewrite (buf0dblwr.cc) : Doublewrite



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

- buffer read entry
  - buf/buf0buf.cc:2491, buf_page_get_gen()

```
2489 UNIV_INTERN
2490 buf_block_t*
2491 buf_page_get_gen(
2492 /*=============*/
2493   ulint   space,  /*!< in: space id */
2494   ulint   zip_size,/*!< in: compressed page size in bytes
2495         or 0 for uncompressed pages */
2496   ulint   offset, /*!< in: page number */
2497   ulint   rw_latch,/*!< in: RW_S_LATCH, RW_X_LATCH, RW_NO_LATCH */
2498   buf_block_t*  guess,  /*!< in: guessed block or NULL */
2499   ulint   mode, /*!< in: BUF_GET, BUF_GET_IF_IN_POOL,
2500         BUF_PEEK_IF_IN_POOL, BUF_GET_NO_LATCH, or
2501         BUF_GET_IF_IN_POOL_OR_WATCH */
2502   const char* file, /*!< in: file name */
2503   ulint   line, /*!< in: line where called */
2504   mtr_t*    mtr)  /*!< in: mini-transaction */
2505 {
2506   buf_block_t*  block;
2507   ulint   fold;
2508   unsigned  access_time;
2509   ulint   fix_type;
2510   rw_lock_t*  hash_lock;
2511   ulint   retries = 0;
2512   buf_block_t*  fix_block;
2513   ib_mutex_t* fix_mutex = NULL;
2514   buf_pool_t* buf_pool = buf_pool_get(space, offset);
```

get the buffer pool ptr using space and offset

* 2 important things

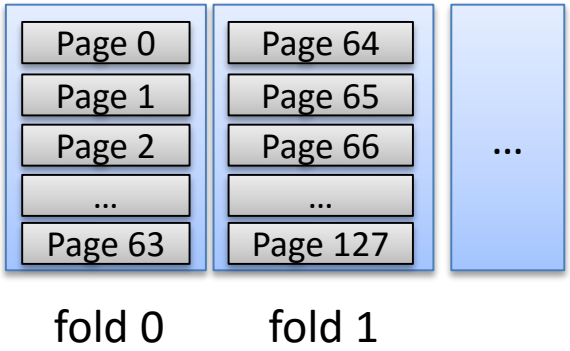
1) ID of a page is space, offset of the page
2) Buffer Pool - Page mapping is mapped

Buffer Read - pool get

- include/buf0buf.ic:1097, buf_pool_get()

```
1092 /*******************************************************************//**
1093 Returns the buffer pool instance given space and offset of page
1094 @return buffer pool */
1095 UNIV_INLINE
1096 buf_pool_t*
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(extent no)



Q : Why fold ?
A : They want to put buffers together in the same buffer pool if it is the same extents for read ahead

Buffer Read

- buf/buf0buf.cc:2491, buf_page_get_gen()

```
2516   ut_ad(mtr);
2517   ut_ad(mtr->state == MTR_ACTIVE);
2518   ut_ad((rw_latch == RW_S_LATCH)
2519         || (rw_latch == RW_X_LATCH)
2520         || (rw_latch == RW_NO_LATCH));
2521 #ifdef UNIV_DEBUG
2522   switch (mode) {
2523   case BUF_GET_NO_LATCH:
2524     ut_ad(rw_latch == RW_NO_LATCH);
2525     break;
2526   case BUF_GET:
2527   case BUF_GET_IF_IN_POOL:
2528   case BUF_PEEK_IF_IN_POOL:
2529   case BUF_GET_IF_IN_POOL_OR_WATCH:
2530   case BUF_GET_POSSIBLY_FREED:
2531     break;
2532   default:
2533     ut_error;
2534   }
2535 #endif /* UNIV_DEBUG */
```

Buffer Read

- buf/buf0buf.cc:2491, buf_page_get_gen()

```
2536   ut_ad(zip_size == fil_space_get_zip_size(space));
2537   ut_ad(ut_is_2pow(zip_size));
2538 #ifndef UNIV_LOG_DEBUG
2539   ut_ad(!ibuf_inside(mtr)
2540         || ibuf_page_low(space, zip_size, offset,
2541                          FALSE, file, line, NULL));
2542 #endif
2543   buf_pool->stat.n_page_gets++;
2544   fold = buf_page_address_fold(space, offset);
2545   hash_lock = buf_page_hash_lock_get(buf_pool, fold);
```

assertion variables

incr. buffer pool stats

get page hash lock, before searching in the hash table

```
2032 /** Get appropriate page_hash_lock. */
2033 # define buf_page_hash_lock_get(b, f)     \
2034   hash_get_lock(b->page_hash, f)
```

Buffer Read - get hash lock

- include/hash0hash.ic:210,hash_get_lock()

```
205 /*************************************************************//**
206 Gets the rw_lock for a fold value in a hash table.
207 @return rw_lock */
208 UNIV_INLINE
209 rw_lock_t*
210 hash_get_lock(
211 /*==========*/
212   hash_table_t* table,  /*!< in: hash table */
213   ulint   fold) /*!< in: fold */
214 {
215   ulint i;
216
217   ut_ad(table);
218   ut_ad(table->type == HASH_TABLE_SYNC_RW_LOCK);
219   ut_ad(table->magic_n == HASH_TABLE_MAGIC_N);
220
221   i = hash_get_sync_obj_index(table, fold);
222
223   return(hash_get_nth_lock(table, i));
224 }
```

get sync obj index

get n-th lock entry

Buffer Read - get hash lock

- include/hash0hash.ic:192,hash_get_nth_lock()

```
187 /*****************************************************//**
188 Gets the nth rw_lock in a hash table.
189 @return rw_lock */
190 UNIV_INLINE
191 rw_lock_t*
192 hash_get_nth_lock(
193 /*==============*/
194   hash_table_t* table,  /*!< in: hash table */
195   ulint   i)  /*!< in: index of the rw_lock */
196 {
197   ut_ad(table);
198   ut_ad(table->magic_n == HASH_TABLE_MAGIC_N);
199   ut_ad(table->type == HASH_TABLE_SYNC_RW_LOCK);
200   ut_ad(i < table->n_sync_obj);
201
202   return(table->sync_obj.rw_locks + i);
203 }
```

get sync obj index

Buffer Read

- buf/buf0buf.cc:2491, buf_page_get_gen()

```
2543   buf_pool->stat.n_page_gets++;
2544   fold = buf_page_address_fold(space, offset);
2545   hash_lock = buf_page_hash_lock_get(buf_pool, fold);
2546 loop:
2547   block = guess;
2548
2549   rw_lock_s_lock(hash_lock);
2550
2551   if (block != NULL) {
2552
2553     /* If the guess is a compressed page descriptor that
2554     has been allocated by buf_page_alloc_descriptor(),
2555     it may have been freed by buf_relocate(). */
2556
2557     if (!buf_block_is_uncompressed(buf_pool, block)
2558         || offset != block->page.offset
2559         || space != block->page.space
2560         || buf_block_get_state(block) != BUF_BLOCK_FILE_PAGE) {
2561
2562       /* Our guess was bogus or things have changed
2563       since. */
2564       block = guess = NULL;
2565     } else {
2566       ut_ad(!block->page.in_zip_hash);
2567     }
2568   }
```

set shared lock on hash table, This is Spin Lock

Note : InnoDB Spin Lock

- include/sync0rw.ic:339, rw_lock_s_lock()

```
339 rw_lock_s_lock_func(
340 /*================*/
341   rw_lock_t*  lock, /*!< in: pointer to rw-lock */
342   ulint   pass, /*!< in: pass value; != 0, if the lock will
343         be passed to another thread to unlock */
344   const char* file_name,/*!< in: file name where lock requested */
345   ulint   line) /*!< in: line where requested */
346 {
347   /* NOTE: As we do not know the thread ids for threads which have
348   s-locked a latch, and s-lockers will be served only after waiting
349   x-lock requests have been fulfilled, then if this thread already
350   owns an s-lock here, it may end up in a deadlock with another thread
351   which requests an x-lock here. Therefore, we will forbid recursive
352   s-locking of a latch: the following assert will warn the programmer
353   of the possibility of this kind of a deadlock. If we want to implement
354   safe recursive s-locking, we should keep in a list the thread ids of
355   the threads which have s-locked a latch. This would use some CPU
356   time. */
357
358 #ifdef UNIV_SYNC_DEBUG
359   ut_ad(!rw_lock_own(lock, RW_LOCK_SHARED)); /* see NOTE above */
360   ut_ad(!rw_lock_own(lock, RW_LOCK_EX));
361 #endif /* UNIV_SYNC_DEBUG */
362
363   if (rw_lock_s_lock_low(lock, pass, file_name, line)) {
364
365     return; /* Success */
366   } else {
367     /* Did not succeed, try spin wait */
368
369     rw_lock_s_lock_spin(lock, pass, file_name, line);
370
371     return;
372   }
373 }
```

read lock word

spin wait

Buffer Read

- buf/buf0buf.cc:2491, buf_page_get_gen()

```
2543   buf_pool->stat.n_page_gets++;
2544   fold = buf_page_address_fold(space, offset);
2545   hash_lock = buf_page_hash_lock_get(buf_pool, fold);
2546 loop:
2547   block = guess;
2548
2549   rw_lock_s_lock(hash_lock);
2550
2551   if (block != NULL) {
2552
2553     /* If the guess is a compressed page descriptor that
2554     has been allocated by buf_page_alloc_descriptor(),
2555     it may have been freed by buf_relocate(). */
2556
2557     if (!buf_block_is_uncompressed(buf_pool, block)
2558         || offset != block->page.offset
2559         || space != block->page.space
2560         || buf_block_get_state(block) != BUF_BLOCK_FILE_PAGE) {
2561
2562       /* Our guess was bogus or things have changed
2563       since. */
2564       block = guess = NULL;
2565     } else {
2566       ut_ad(!block->page.in_zip_hash);
2567     }
2568   }
```

we have a page hash

Buffer Read

- buf/buf0buf.cc:2491, buf_page_get_gen()

```
2570  if (block == NULL) {
2571    block = (buf_block_t*) buf_page_hash_get_low(
2572      buf_pool, space, offset, fold);
2573  }
2574
2575  if (!block || buf_pool_watch_is_sentinel(buf_pool, &block->page)) {
2576    rw_lock_s_unlock(hash_lock);
2577    block = NULL;
2578  }
2579
2580  if (block == NULL) {
2581    /* Page not in buf_pool: needs to be read from file */
2582
2583    if (mode == BUF_GET_IF_IN_POOL_OR_WATCH) {
2584      rw_lock_x_lock(hash_lock);
2585      block = (buf_block_t*) buf_pool_watch_set(
2586        space, offset, fold);
2587
2588      if (UNIV_LIKELY_NULL(block)) {
2589        /* We can release hash_lock after we
2590        increment the fix count to make
2591        sure that no state change takes place. */
2592        fix_block = block;
2593        buf_block_fix(fix_block);
2594
2595        /* Now safe to release page_hash mutex */
2596        rw_lock_x_unlock(hash_lock);
2597        goto got_block;
2598      }
```

find a page in the hash table

Buffer Read - hash search

- include/buf0buf.ic:1132, buf_page_hash_get_low()

```
1130 UNIV_INLINE
1131 buf_page_t*
1132 buf_page_hash_get_low(
1133 /*==================*/
1134   buf_pool_t* buf_pool,/*!< buffer pool instance */
1135   ulint   space,  /*!< in: space id */
1136   ulint   offset, /*!< in: offset of the page within space */
1137   ulint   fold) /*!< in: buf_page_address_fold(space, offset) */
1138 {
1139   buf_page_t* bpage;
1140
1141 #ifdef UNIV_SYNC_DEBUG
1142   ulint   hash_fold;
1143   rw_lock_t*  hash_lock;
1144
1145   hash_fold = buf_page_address_fold(space, offset);
1146   ut_ad(hash_fold == fold);
1147
1148   hash_lock = hash_get_lock(buf_pool->page_hash, fold);
1149   ut_ad(rw_lock_own(hash_lock, RW_LOCK_EX)
1150         || rw_lock_own(hash_lock, RW_LOCK_SHARED));
1151 #endif /* UNIV_SYNC_DEBUG */
1152
1153   /* Look for the page in the hash table */
1154
1155   HASH_SEARCH(hash, buf_pool->page_hash, fold, buf_page_t*, bpage,
1156         ut_ad(bpage->in_page_hash && !bpage->in_zip_hash
1157         && buf_page_in_file(bpage)),
1158         bpage->space == space && bpage->offset == offset);
1159   if (bpage) {
1160     ut_a(buf_page_in_file(bpage));
1161     ut_ad(bpage->in_page_hash);
1162     ut_ad(!bpage->in_zip_hash);
1163   }
1164
1165   return(bpage);
1166 }
```

search hash table

found!!

Page hash - hash table

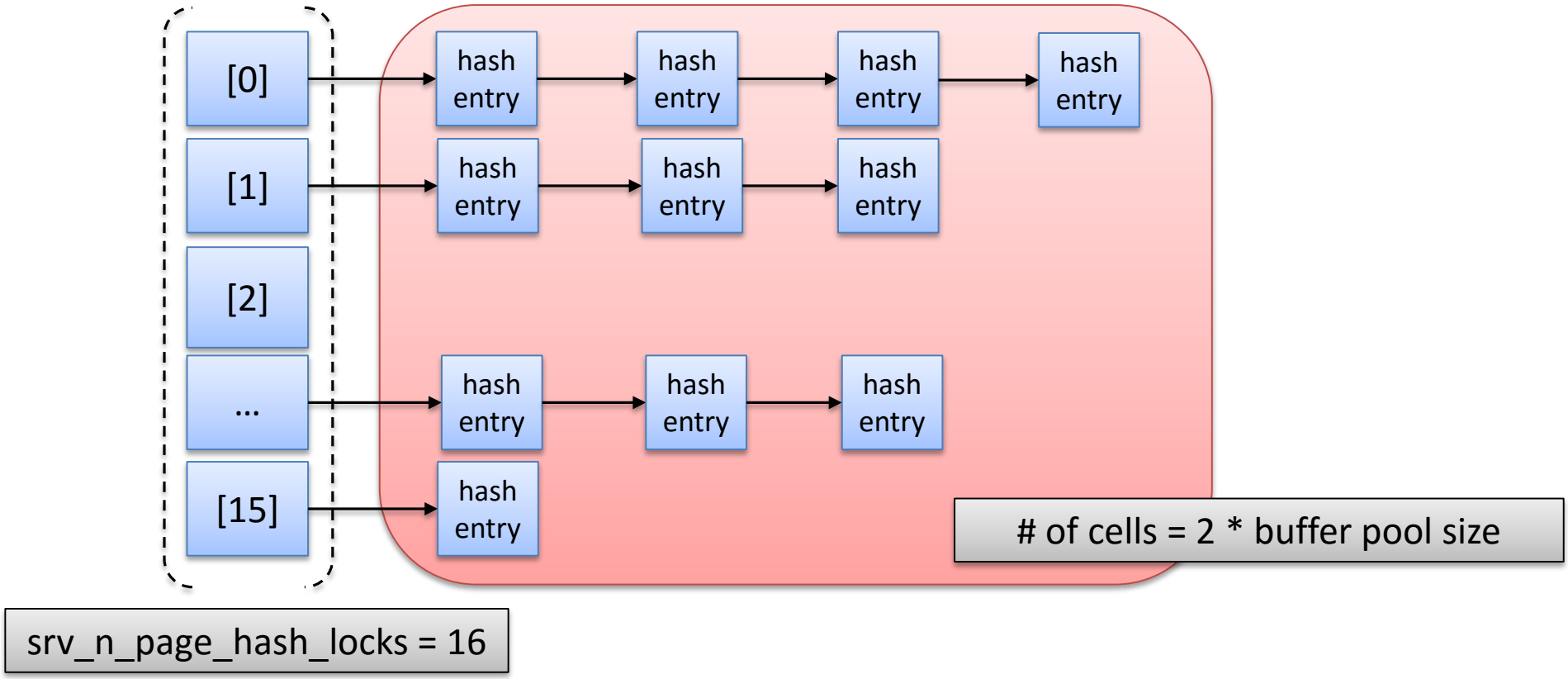
- include/hash0hash.h:532, hash_table_t

```
531 /* The hash table structure */
532 struct hash_table_t {
533   enum hash_table_sync_t  type; /*<! type of hash_table. */
534 #if defined UNIV_AHI_DEBUG || defined UNIV_DEBUG
535 # ifndef UNIV_HOTBACKUP
536   ibool     adaptive;/* TRUE if this is the hash
537           table of the adaptive hash
538           index */
539 # endif /* !UNIV_HOTBACKUP */
540 #endif /* UNIV_AHI_DEBUG || UNIV_DEBUG */
541   ulint     n_cells;/* number of cells in the hash table */
542   hash_cell_t*    array;  /*!< pointer to cell array */
543 #ifndef UNIV_HOTBACKUP
544   ulint     n_sync_obj;/* if sync_objs != NULL, then
545           the number of either the number
546           of mutexes or the number of
547           rw_locks depending on the type.
548           Must be a power of 2 */
549   union {
550     ib_mutex_t* mutexes;/* NULL, or an array of mutexes
551           used to protect segments of the
552           hash table */
553     rw_lock_t*  rw_locks;/* NULL, or an array of rw_lcoks
554           used to protect segments of the
555           hash table */
556   } sync_obj;
557
558   mem_heap_t**    heaps;  /*!< if this is non-NULL, hash
559           chain nodes for external chaining
560           can be allocated from these memory
561           heaps; there are then n_mutexes
562           many of these heaps */
563 #endif /* !UNIV_HOTBACKUP */
564   mem_heap_t*   heap;
565 #ifdef UNIV_DEBUG
566   ulint     magic_n;
567 # define HASH_TABLE_MAGIC_N 76561114
568 #endif /* UNIV_DEBUG */
569 };
```

Page hash - hash table

- Hash table

buf_pool_t->page_hash



Buffer Read - hash search

- include/hash0hash.h:197, HASH_SEARCH

```
1155   HASH_SEARCH(hash, buf_pool->page_hash, fold, buf_page_t*, bpage,
1156         ut_ad(bpage->in_page_hash && !bpage->in_zip_hash
1157         && buf_page_in_file(bpage)),
1158         bpage->space == space && bpage->offset == offset);
```

```
195 /********************************************************************//**
196 Looks for a struct in a hash table. */
197 #define HASH_SEARCH(NAME, TABLE, FOLD, TYPE, DATA, ASSERTION, TEST)\
198 {\
199 \
200   HASH_ASSERT_OWN(TABLE, FOLD)\
201 \
202   (DATA) = (TYPE) HASH_GET_FIRST(TABLE, hash_calc_hash(FOLD, TABLE));\
203   HASH_ASSERT_VALID(DATA);\
204 \
205   while ((DATA) != NULL) {\
206     ASSERTION;\
207     if (TEST) {\
208       break;\
209     } else {\
210       HASH_ASSERT_VALID(HASH_GET_NEXT(NAME, DATA));\
211       (DATA) = (TYPE) HASH_GET_NEXT(NAME, DATA);\
212     }\
213   }\
214 }
```

get the first hash entry

if the current list is not end

is this what we looking for ?

if this is not what we looking for, then get the next entry

Buffer Read

- buf/buf0buf.cc:2491, buf_page_get_gen()

```
2570  if (block == NULL) {
2571    block = (buf_block_t*) buf_page_hash_get_low(
2572      buf_pool, space, offset, fold);
2573  }
2574
2575  if (!block || buf_pool_watch_is_sentinel(buf_pool, &block->page)) {
2576    rw_lock_s_unlock(hash_lock);
2577    block = NULL;
2578  }
2579
2580  if (block == NULL) {
2581    /* Page not in buf_pool: needs to be read from file */
2582
2583    if (mode == BUF_GET_IF_IN_POOL_OR_WATCH) {
2584      rw_lock_x_lock(hash_lock);
2585      block = (buf_block_t*) buf_pool_watch_set(
2586        space, offset, fold);
2587
2588      if (UNIV_LIKELY_NULL(block)) {
2589        /* We can release hash_lock after we
2590        increment the fix count to make
2591        sure that no state change takes place. */
2592        fix_block = block;
2593        buf_block_fix(fix_block);
2594
2595        /* Now safe to release page_hash mutex */
2596        rw_lock_x_unlock(hash_lock);
2597        goto got_block;
2598      }
```

check current control block is sentinel : details later

if current block is NULL, then we couldn't find a requested page

Buffer Read

- buf/buf0buf.cc:2491, buf_page_get_gen()

```
2613    if (buf_read_page(space, zip_size, offset)) {
2614      buf_read_ahead_random(space, zip_size, offset,
2615                ibuf_inside(mtr));
2616
2617      retries = 0;
2618    } else if (retries < BUF_PAGE_READ_MAX_RETRIES) {
2619      ++retries;
2620      DBUG_EXECUTE_IF(
2621        "innodb_page_corruption_retries",
2622        retries = BUF_PAGE_READ_MAX_RETRIES;
2623      );
2624    } else {
2625      fprintf(stderr, "InnoDB: Error: Unable"
2626        " to read tablespace %lu page no"
2627        " %lu into the buffer pool after"
2628        " %lu attempts\n"
2629        "InnoDB: The most probable cause"
2630        " of this error may be that the"
2631        " table has been corrupted.\n"
2632        "InnoDB: You can try to fix this"
2633        " problem by using"
2634        " innodb_force_recovery.\n"
2635        "InnoDB: Please see reference manual"
2636        " for more details.\n"
2637        "InnoDB: Aborting...\n",
2638        space, offset,
2639        BUF_PAGE_READ_MAX_RETRIES);
2640
2641      ut_error;
2642    }
```

read success

currently block is NULL and read a page

fail but within retry case

read fail case

Buffer Read - buf_read_page()

- buf/buf0rea.cc:394, buf_read_page()

```
392 UNIV_INTERN
393 ibool
394 buf_read_page(
395 /*==========*/
396   ulint space,  /*!< in: space id */
397   ulint zip_size,/*!< in: compressed page size in bytes, or 0 */
398   ulint offset) /*!< in: page number */
399 {
400   ib_int64_t  tablespace_version;
401   ulint   count;
402   dberr_t   err;
403
404   tablespace_version = fil_space_get_version(space);
405
406   /* We do the i/o in the synchronous aio mode to save thread
407   switches: hence TRUE */
408
409   count = buf_read_page_low(&err, true, BUF_READ_ANY_PAGE, space,
410           zip_size, FALSE,
411           tablespace_version, offset);
412   srv_stats.buf_pool_reads.add(count);
413   if (err == DB_TABLESPACE_DELETED) {
414     ut_print_timestamp(stderr);
415     fprintf(stderr,
416       "  InnoDB: Error: trying to access"
417       " tablespace %lu page no. %lu,\n"
418       "InnoDB: but the tablespace does not exist"
419       " or is just being dropped.\n",
420       (ulong) space, (ulong) offset);
421   }
422
423   /* Increment number of I/O operations used for LRU policy. */
424   buf_LRU_stat_inc_io();
425
426   return(count > 0);
427 }
```

read_page_low

increase LRU stat

Buffer Read - buf_read_page()

- buf/buf0rea.cc:105, buf_read_page_low()

```
103 static
104 ulint
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;
126   ibool   ignore_nonexistent_pages;
127
128   *err = DB_SUCCESS;
129
130   wake_later = mode & OS_AIO_SIMULATED_WAKE_LATER;
131   mode = mode & ~OS_AIO_SIMULATED_WAKE_LATER;
```

sync read ? **true**

read ahead mode

Buffer Read - buf_read_page()

- buf/buf0rea.cc:105, buf_read_page_low()

```
133   ignore_nonexistent_pages = mode & BUF_READ_IGNORE_NONEXISTENT_PAGES;
134   mode &= ~BUF_READ_IGNORE_NONEXISTENT_PAGES;
135
136   if (space == TRX_SYS_SPACE && buf_dblwr_page_inside(offset)) {
137     ut_print_timestamp(stderr);
138     fprintf(stderr,
139       "  InnoDB: Warning: trying to read"
140       " doublewrite buffer page %lu\n",
141       (ulong) offset);
142
143     return(0);
144   }
145
146   if (ibuf_bitmap_page(zip_size, offset)
147       || trx_sys_hdr_page(space, offset)) {
148
149     /* Trx sys header is so low in the latching order that we play
150     safe and do not leave the i/o-completion to an asynchronous
151     i/o-thread. Ibuf bitmap pages must always be read with
152     syncronous i/o, to make sure they do not get involved in
153     thread deadlocks. */
154
155     sync = true;
156   }
```

read double write page ?

read ibuf page ?

Buffer Read - buf_read_page()

- buf/buf0rea.cc:105, 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   }
168
169 #ifdef UNIV_DEBUG
170   if (buf_debug_prints) {
171     fprintf(stderr,
172       "Posting read request for page %lu, sync %s\n",
173       (ulong) offset, sync ? "true" : "false");
174   }
175 #endif
176
177   ut_ad(buf_page_in_file(bpage));
178
179   if (sync) {
180     thd_wait_begin(NULL, THD_WAIT_DISKIO);
181   }
```

allocate buffer block for read

if it was sync read ? then wait until it done.

Buffer Read - buf_read_page()

- buf/buf0buf.cc:3477, buf_page_init_for_read()

```
3475 UNIV_INTERN
3476 buf_page_t*
3477 buf_page_init_for_read(
3478 /*=====================*/
3479   dberr_t*  err,  /*!< out: DB_SUCCESS or DB_TABLESPACE_DELETED */
3480   ulint   mode, /*!< in: BUF_READ_IBUF_PAGES_ONLY, ... */
3481   ulint   space,  /*!< in: space id */
3482   ulint   zip_size,/*!< in: compressed page size, or 0 */
3483   ibool   unzip,  /*!< in: TRUE=request uncompressed page */
3484   ib_int64_t  tablespace_version,
3485         /*!< in: prevents reading from a wrong
3486         version of the tablespace in case we have done
3487         DISCARD + IMPORT */
3488   ulint   offset) /*!< in: page number */
3489 {
3490   buf_block_t*  block;
3491   buf_page_t* bpage = NULL;
3492   buf_page_t* watch_page;
3493   rw_lock_t*  hash_lock;
3494   mtr_t   mtr;
3495   ulint   fold;
3496   ibool   lru = FALSE;
3497   void*   data;
3498   buf_pool_t* buf_pool = buf_pool_get(space, offset);
3499
3500   ut_ad(buf_pool);
3501
3502   *err = DB_SUCCESS;
```

Buffer Read - buf_read_page()

- buf/buf0buf.cc:3477, buf_page_init_for_read()

```
3504   if (mode == BUF_READ_IBUF_PAGES_ONLY) {
3505     /* It is a read-ahead within an ibuf routine */
3506
3507     ut_ad(!ibuf_bitmap_page(zip_size, offset));
3508
3509     ibuf_mtr_start(&mtr);
3510
3511     if (!recv_no_ibuf_operations
3512         && !ibuf_page(space, zip_size, offset, &mtr)) {
3513
3514       ibuf_mtr_commit(&mtr);
3515
3516       return(NULL);
3517     }
3518   } else {
3519     ut_ad(mode == BUF_READ_ANY_PAGE);
3520   }
```

check read mode

Buffer Read - buf_read_page()

- buf/buf0buf.cc:3477, buf_page_init_for_read()

```
3522   if (zip_size && !unzip && !recv_recovery_is_on()) {
3523     block = NULL;
3524   } else {
3525     block = buf_LRU_get_free_block(buf_pool);
3526     ut_ad(block);
3527     ut_ad(buf_pool_from_block(block) == buf_pool);
3528   }
3529
3530   fold = buf_page_address_fold(space, offset);
3531   hash_lock = buf_page_hash_lock_get(buf_pool, fold);
3532
3533   buf_pool_mutex_enter(buf_pool);
3534   rw_lock_x_lock(hash_lock);
3535
```

get free block: see this later

Buffer Read - buf_read_page()

- buf/buf0buf.cc:3477, buf_page_init_for_read()

```
3536   watch_page = buf_page_hash_get_low(buf_pool, space, offset, fold);
3537   if (watch_page && !buf_pool_watch_is_sentinel(buf_pool, watch_page)) {
3538     /* The page is already in the buffer pool. */
3539     watch_page = NULL;
3540 err_exit:
3541     rw_lock_x_unlock(hash_lock);
3542     if (block) {
3543       mutex_enter(&block->mutex);
3544       buf_LRU_block_free_non_file_page(block);
3545       mutex_exit(&block->mutex);
3546     }
3547
3548     bpage = NULL;
3549     goto func_exit;
3550   }
3551
3552   if (fil_tablespace_deleted_or_being_deleted_in_mem(
3553         space, tablespace_version)) {
3554     /* The page belongs to a space which has been
3555     deleted or is being deleted. */
3556     *err = DB_TABLESPACE_DELETED;
3557
3558     goto err_exit;
3559   }
```

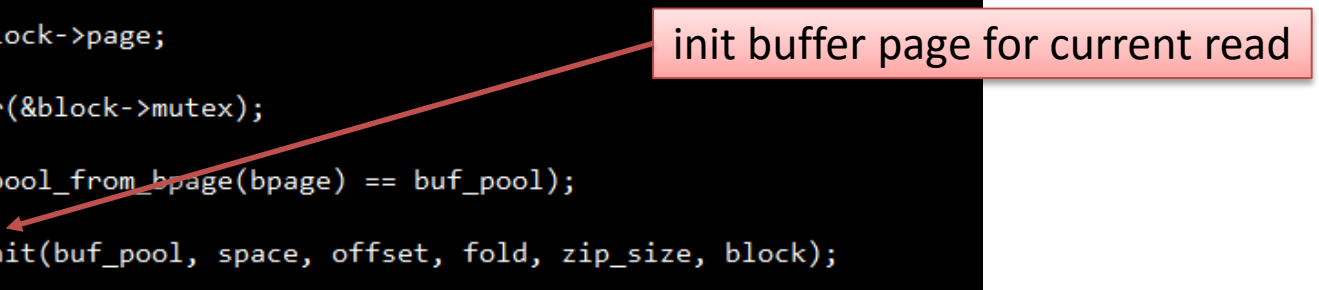
other thread already get my block

deleted error handling

Buffer Read - buf_read_page()

- buf/buf0buf.cc:3477, buf_page_init_for_read()

```
3561   if (block) {
3562     bpage = &block->page;
3563
3564     mutex_enter(&block->mutex);
3565
3566     ut_ad(buf_pool_from_bpage(bpage) == buf_pool);
3567
3568     buf_page_init(buf_pool, space, offset, fold, zip_size, block);
3569
3570 #ifdef PAGE_ATOMIC_REF_COUNT
3571     /* Note: We set the io state without the protection of
3572     the block->lock. This is because other threads cannot
3573     access this block unless it is in the hash table. */
3574
3575     buf_page_set_io_fix(bpage, BUF_IO_READ);
3576 #endif /* PAGE_ATOMIC_REF_COUNT */
3577
3578     rw_lock_x_unlock(hash_lock);
3579
3580     /* The block must be put to the LRU list, to the old blocks */
3581     buf_LRU_add_block(bpage, TRUE/* to old blocks */);
```



Buffer Read - buf_read_page()

- buf/buf0buf.cc:3376, buf_page_init()

```
3374 static __attribute__((nonnull))
3375 void
3376 buf_page_init(
3377 /*==========*/
3378   buf_pool_t* buf_pool,/*!< in/out: buffer pool */
3379   ulint   space,  /*!< in: space id */
3380   ulint   offset, /*!< in: offset of the page within space
3381         in units of a page */
3382   ulint   fold, /*!< in: buf_page_address_fold(space,offset) */
3383   ulint   zip_size,/*!< in: compressed page size, or 0 */
3384   buf_block_t*  block)  /*!< in/out: block to init */
3385 {
3386   buf_page_t* hash_page;
3387
3388   ut_ad(buf_pool == buf_pool_get(space, offset));
3389   ut_ad(buf_pool_mutex_own(buf_pool));
3390
3391   ut_ad(mutex_own(&(block->mutex)));
3392   ut_a(buf_block_get_state(block) != BUF_BLOCK_FILE_PAGE);
3393
3394 #ifdef UNIV_SYNC_DEBUG
3395   ut_ad(rw_lock_own(buf_page_hash_lock_get(buf_pool, fold),
3396         RW_LOCK_EX));
3397 #endif /* UNIV_SYNC_DEBUG */
```

Do assert

Buffer Read - buf_read_page()

- buf/buf0buf.cc:3376, buf_page_init()

```
3399   /* Set the state of the block */
3400   buf_block_set_file_page(block, space, offset);
3401
3402 #ifdef UNIV_DEBUG_VALGRIND
3403   if (!space) {
3404     /* Silence valid Valgrind warnings about uninitialized
3405     data being written to data files.  There are some unused
3406     bytes on some pages that InnoDB does not initialize. */
3407     UNIV_MEM_VALID(block->frame, UNIV_PAGE_SIZE);
3408   }
3409 #endif /* UNIV_DEBUG_VALGRIND */
3410
3411   buf_block_init_low(block);
3412
3413   block->lock_hash_val = lock_rec_hash(space, offset);
3414
3415   buf_page_init_low(&block->page);
```

```
397 /*********************************************************************//**
398 Map a block to a file page. */
399 UNIV_INLINE
400 void
401 buf_block_set_file_page(
402 /*====================*/
403   buf_block_t*    block,  /*!< in/out: pointer to control block */
404   ulint     space,  /*!< in: tablespace id */
405   ulint     page_no)/*!< in: page number */
406 {
407   buf_block_set_state(block, BUF_BLOCK_FILE_PAGE);
408   block->page.space = static_cast<ib_uint32_t>(space);
409   block->page.offset = static_cast<ib_uint32_t>(page_no);
410 }
```

Buffer Read - buf_read_page()

- buf/buf0buf.cc:3376, buf_page_init()

```
3399   /* Set the state of the block */
3400   buf_block_set_file_page(block, space, offset);
3401
3402 #ifdef UNIV_DEBUG_VALGRIND
3403   if (!space) {
3404     /* Silence valid Valgrind warnings about uninitialized
3405     data being written to data files.  There are some unused
3406     bytes on some pages that InnoDB does not initialize. */
3407     UNIV_MEM_VALID(block->frame, UNIV_PAGE_SIZE);
3408   }
3409 #endif /* UNIV_DEBUG_VALGRIND */
3410
3411   buf_block_init_low(block);
3412
3413   block->lock_hash_val = lock_rec_hash(space, offset);
3414
3415   buf_page_init_low(&block->page);
```

```
2145 /*********************************************************************//**
2146 Initialize some fields of a control block. */
2147 UNIV_INLINE
2148 void
2149 buf_block_init_low(
2150 /*===============*/
2151   buf_block_t*  block)  /*!< in: block to init */
2152 {
2153   block->check_index_page_at_flush = FALSE;
2154   block->index     = NULL;
2155
2156   block->n_hash_helps = 0;
2157   block->n_fields   = 1;
2158   block->n_bytes    = 0;
2159   block->left_side  = TRUE;
2160 }
```

Buffer Read - buf_read_page()

- buf/buf0buf.cc:3376, buf_page_init()

```
3399   /* Set the state of the block */
3400   buf_block_set_file_page(block, space, offset);
3401
3402 #ifdef UNIV_DEBUG_VALGRIND
3403   if (!space) {
3404     /* Silence valid Valgrind warnings about uninitialized
3405     data being written to data files.  There are some unused
3406     bytes on some pages that InnoDB does not initialize. */
3407     UNIV_MEM_VALID(block->frame, UNIV_PAGE_SIZE);
3408   }
3409 #endif /* UNIV_DEBUG_VALGRIND */
3410
3411   buf_block_init_low(block);
3412
3413   block->lock_hash_val = lock_rec_hash(sp
3414
3415   buf_page_init_low(&block->page);
```

```
3351 /********************************************************************//**
3352 Initialize some fields of a control block. */
3353 UNIV_INLINE
3354 void
3355 buf_page_init_low(
3356 /*==============*/
3357   buf_page_t* bpage)  /*!< in: block to init */
3358 {
3359   bpage->flush_type = BUF_FLUSH_LRU;
3360   bpage->io_fix = BUF_IO_NONE;
3361   bpage->buf_fix_count = 0;
3362   bpage->freed_page_clock = 0;
3363   bpage->access_time = 0;
3364   bpage->newest_modification = 0;
3365   bpage->oldest_modification = 0;
3366   HASH_INVALIDATE(bpage, hash);
3367 #if defined UNIV_DEBUG_FILE_ACCESSES || defined UNIV_DEBUG
3368   bpage->file_page_was_freed = FALSE;
3369 #endif /* UNIV_DEBUG_FILE_ACCESSES || UNIV_DEBUG */
3370 }
```

Buffer Read - buf_read_page()

- buf/buf0buf.cc:3376, buf_page_init()

```
3417   /* Insert into the hash table of file pages */
3418
3419   hash_page = buf_page_hash_get_low(buf_pool, space, offset, fold);
3420
3421   if (hash_page == NULL) {
3422     /* Block not found in the hash tal
3423   } else if (buf_pool_watch_is_sentinel(buf_pool, hash_page)) {
3424     ib_uint32_t buf_fix_count = hash_page->buf_fix_count;
3425
3426   ut_a(buf_fix_count > 0);
3427
3428 #ifdef PAGE_ATOMIC_REF_COUNT
3429     os_atomic_increment_uint32(
3430       &block->page.buf_fix_count, buf_fix_count);
3431 #else
3432     block->page.buf_fix_count += ulint(buf_fix_count);
3433 #endif /* PAGE_ATOMIC_REF_COUNT */
3434
3435     buf_pool_watch_remove(buf_pool, fold, hash_page);
3436   } else {
3437     fprintf(stderr,
3438       "InnoDB: Error: page %lu %lu already found"
3439       " in the hash table: %p, %p\n",
3440       (ulong) space,
3441       (ulong) offset,
3442       (const void*) hash_page, (const void*) block);
3443 #if defined UNIV_DEBUG || defined UNIV_BUF_DEBUG
3444     mutex_exit(&block->mutex);
3445     buf_pool_mutex_exit(buf_pool);
3446     buf_print();
3447     buf_LRU_print();
3448     buf_validate();
3449     buf_LRU_validate();
3450 #endif /* UNIV_DEBUG || UNIV_BUF_DEBUG */
3451     ut_error;
3452   }
```

need to insert a page into hash table

check already allocate a buf page

error handling

Buffer Read - buf_read_page()

- buf/buf0buf.cc:3376, buf_page_init()

```
3453
3454   ut_ad(!block->page.in_zip_hash);
3455   ut_ad(!block->page.in_page_hash);
3456   ut_d(block->page.in_page_hash = TRUE);
3457
3458   HASH_INSERT(buf_page_t, hash, buf_pool->page_hash, fold, &block->page);
3459
3460   if (zip_size) {
3461     page_zip_set_size(&block->page.zip,
3462   }
3463 }
```

hash insert

```
119 /*******************************************************************//**
120 Inserts a struct to a hash table. */
121
122 #define HASH_INSERT(TYPE, NAME, TABLE, FOLD, DATA)\
123 do {\
124   hash_cell_t*  cell3333;\
125   TYPE*   struct3333;\
126 \
127   HASH_ASSERT_OWN(TABLE, FOLD)\
128 \
129   (DATA)->NAME = NULL;\
130 \
131   cell3333 = hash_get_nth_cell(TABLE, hash_calc_hash(FOLD, TABLE));\
132 \
133   if (cell3333->node == NULL) {\
134     cell3333->node = DATA;\
135   } else {\
136     struct3333 = (TYPE*) cell3333->node;\
137 \
138     while (struct3333->NAME != NULL) {\
139 \
140       struct3333 = (TYPE*) struct3333->NAME;\
141     }\
142 \
143     struct3333->NAME = DATA;\
144   }\
145 } while (0)
```

Buffer Read - buf_read_page()

- buf/buf0buf.cc:3477, buf_page_init_for_read()

```
3561   if (block) {
3562     bpage = &block->page;
3563
3564     mutex_enter(&block->mutex);
3565
3566     ut_ad(buf_pool_from_bpage(bpage) == buf_pool);
3567
3568     buf_page_init(buf_pool, space, offset, fold, zip_size, block);
3569
3570 #ifdef PAGE_ATOMIC_REF_COUNT
3571     /* Note: We set the io state without the protection of
3572     the block->lock. This is because other threads cannot
3573     access this block unless it is in the hash table. */
3574
3575     buf_page_set_io_fix(bpage, BUF_IO_READ);
3576 #endif /* PAGE_ATOMIC_REF_COUNT */
3577
3578     rw_lock_x_unlock(hash_lock);
3579
3580     /* The block must be put to the LRU list, to the old blocks */
3581     buf_LRU_add_block(bpage, TRUE/* to old blocks */);
```

set io fix BUF_IO_READ

add current block to LRU list : see this later

Buffer Read - buf_read_page()

- buf/buf0buf.cc:3477, buf_page_init_for_read()

```
3721   buf_pool->n_pend_reads++;
3722 func_exit:
3723   buf_pool_mutex_exit(buf_pool);
3724
3725   if (mode == BUF_READ_IBUF_PAGES_ONLY) {
3726
3727     ibuf_mtr_commit(&mtr);
3728   }
3729
3730
3731 #ifdef UNIV_SYNC_DEBUG
3732   ut_ad(!rw_lock_own(hash_lock, RW_LOCK_EX));
3733   ut_ad(!rw_lock_own(hash_lock, RW_LOCK_SHARED));
3734 #endif /* UNIV_SYNC_DEBUG */
3735
3736   ut_ad(!bpage || buf_page_in_file(bpage));
3737   return(bpage);
3738 }
```

Increase pending read count : how many buffer read were requested and not finished

now we allocate a free buffer and control block, and it was inserted into hash table and LRU list

Buffer Read - buf_read_page()

- buf/buf0rea.cc:105, buf_read_page_low()

```
183   if (zip_size) {
184     *err = fil_io(OS_FILE_READ | wake_later
185             | ignore_nonexistent_pages,
186             sync, space, zip_size, offset, 0, zip_size,
187             bpage->zip.data, bpage);
188   } else {
189     ut_a(buf_page_get_state(bpage) == BUF_BLOCK_FILE_PAGE);
190
191     *err = fil_io(OS_FILE_READ | wake_later
192             | ignore_nonexistent_pages,
193             sync, space, 0, offset, 0, UNIV_PAGE_SIZE,
194             ((buf_block_t*) bpage)->frame, bpage);
195   }
196
197   if (sync) {
198     thd_wait_end(NULL);
199   }
```

Buffer Read - buf_read_page()

- buf/buf0rea.cc:105, buf_read_page_low()

```
201   if (*err != DB_SUCCESS) {
202     if (ignore_nonexistent_pages || *err == DB_TABLESPACE_DELETED) {
203       buf_read_page_handle_error(bpage);
204       return(0);
205     }
206     /* else */
207     ut_error;
208   }
209
210   if (sync) {
211     /* The i/o is already completed when we arrive from
212     fil_read */
213     if (!buf_page_io_complete(bpage)) {
214       return(0);
215     }
216   }
217
218   return(1);
219 }
```

sync read, do io complete

Buffer Read - buf_page_io_complete()

- buf/buf0rea.cc:4051, buf_page_io_complete()

```
4049 UNIV_INTERN
4050 bool
4051 buf_page_io_complete(
4052 /*=====================*/
4053   buf_page_t* bpage)  /*!< in: pointer to the block in question */
4054 {
4055   enum buf_io_fix io_type;
4056   buf_pool_t* buf_pool = buf_pool_from_bpage(bpage);
4057   const ibool uncompressed = (buf_page_get_state(bpage)
4058           == BUF_BLOCK_FILE_PAGE);
4059
4060   ut_a(buf_page_in_file(bpage));
4061
4062   /* We do not need protect io_fix here by mutex to read
4063   it because this is the only function where we can change the value
4064   from BUF_IO_READ or BUF_IO_WRITE to some other value, and our code
4065   ensures that this is the only thread that handles the i/o for this
4066   block. */
4067
4068   io_type = buf_page_get_io_fix(bpage);
4069   ut_ad(io_type == BUF_IO_READ || io_type == BUF_IO_WRITE);
4070
4071   if (io_type == BUF_IO_READ) {
4072     ulint read_page_no;
4073     ulint read_space_id;
4074     byte* frame;
```

Buffer Read - buf_page_io_complete()

- buf/buf0rea.cc:4051, buf_page_io_complete()

```
4129     /* From version 3.23.38 up we store the page checksum
4130     to the 4 first bytes of the page end lsn field */
4131
4132     if (buf_page_is_corrupted(true, frame,
4133             buf_page_get_zip_size(bpage))) {
4134
4135       /* Not a real corruption if it was triggered by
4136       error injection */
4137       DBUG_EXECUTE_IF("buf_page_is_corrupt_failure",
4138         if (bpage->space > TRX_SYS_SPACE
4139             && buf_mark_space_corrupt(bpage)) {
4140           ib_logf(IB_LOG_LEVEL_INFO,
4141             "Simulated page corruption");
4142           return(true);
4143         }
4144         goto page_not_corrupt;
4145         ;);
4146 corrupt:
4147       fprintf(stderr,
4148         "InnoDB: Database page corruption on disk"
4149         " or a failed\n"
4150         "InnoDB: file read of page %lu.\n"
4151         "InnoDB: You may have to recover"
4152         " from a backup.\n",
4153         (ulong) bpage->offset);
4154       buf_page_print(frame, buf_page_get_zip_size(bpage),
4155               BUF_PAGE_PRINT_NO_CRASH);
```

page corruption check based on checksum in the page : see this later

Buffer Read - buf_page_io_complete()

- buf/buf0rea.cc:4051, buf_page_io_complete()

```
4218   buf_pool_mutex_enter(buf_pool);
4219   mutex_enter(buf_page_get_mutex(bpage));
4220
4221 #ifdef UNIV_IBUF_COUNT_DEBUG
4222   if (io_type == BUF_IO_WRITE || uncompressed) {
4223     /* For BUF_IO_READ of compressed-only blocks, the
4224     buffered operations will be merged by buf_page_get_gen()
4225     after the block has been uncompressed. */
4226     ut_a(ibuf_count_get(bpage->space, bpage->offset) == 0);
4227   }
4228 #endif
4229   /* Because this thread which does the unlocking is not the same that
4230   did the locking, we use a pass value != 0 in unlock, which simply
4231   removes the newest lock debug record, without checking the thread
4232   id. */
4233
4234   buf_page_set_io_fix(bpage, BUF_IO_NONE);
```

change IO fix to none

Buffer Read - buf_page_io_complete()

- buf/buf0rea.cc:4051, buf_page_io_complete()

```
4236   switch (io_type) {
4237   case BUF_IO_READ:
4238     /* NOTE that the call to ibuf may have moved the ownership of
4239     the x-latch to this OS thread: do not let this confuse you in
4240     debugging! */
4241
4242     ut_ad(buf_pool->n_pend_reads > 0);
4243     buf_pool->n_pend_reads--;
4244     buf_pool->stat.n_pages_read++;
4245
4246     if (uncompressed) {
4247       rw_lock_x_unlock_gen(&((buf_block_t*) bpage)->lock,
4248                BUF_IO_READ);
4249     }
4250
4251     break;
```

Buffer Read

- buf/buf0buf.cc:2491, buf_page_get_gen()

```
2618     } else if (retries < BUF_PAGE_READ_MAX_RETRIES) {
2619       ++retries;
2620       DBUG_EXECUTE_IF(
2621         "innodb_page_corruption_retries",
2622         retries = BUF_PAGE_READ_MAX_RETRIES;
2623       );
2624     } else {
2625       fprintf(stderr, "InnoDB: Error: Unable"
2626         " to read tablespace %lu page no"
2627         " %lu into the buffer pool after"
2628         " %lu attempts\n"
2629         "InnoDB: The most probable cause"
2630         " of this error may be that the"
2631         " table has been corrupted.\n"
2632         "InnoDB: You can try to fix this"
2633         " problem by using"
2634         " innodb_force_recovery.\n"
2635         "InnoDB: Please see reference manual"
2636         " for more details.\n"
2637         "InnoDB: Aborting...\n",
2638         space, offset,
2639         BUF_PAGE_READ_MAX_RETRIES);
2640
2641       ut_error;
2642     }
2643
2644 #if defined UNIV_DEBUG || defined UNIV_BUF_DEBUG
2645     ut_a(++buf_dbg_counter % 5771 || buf_validate());
2646 #endif /* UNIV_DEBUG || UNIV_BUF_DEBUG */
2647     goto loop;
```

goto loop!!

Buffer Read

- buf/buf0buf.cc:2491, buf_page_get_gen()

```
2648    } else {
2649      fix_block = block;
2650    }
2651
2652    buf_block_fix(fix_block);
2653
2654    /* Now safe to release page_hash mutex */
2655    rw_lock_s_unlock(hash_lock);
2656
2657 got_block:
```

fix current block and unlock hash lock

```
1001 /********************************************************************//**
1002 Increments the bufferfix count. */
1003 UNIV_INLINE
1004 void
1005 buf_block_fix(
1006 /*==========*/
1007   buf_block_t*  block)  /*!< in/out: block to bufferfix */
1008 {
1009 #ifdef PAGE_ATOMIC_REF_COUNT
1010   os_atomic_increment_uint32(&block->page.buf_fix_count, 1);
1011 #else
1012   ib_mutex_t* block_mutex = buf_page_get_mutex(&block->page);
1013
1014   mutex_enter(block_mutex);
1015   ++block->page.buf_fix_count;
1016   mutex_exit(block_mutex);
1017 #endif /* PAGE_ATOMIC_REF_COUNT */
1018 }
```

READ A PAGE - AFTER GOT BLOCK

Buffer Read

- buf/buf0buf.cc:2491, buf_page_get_gen()

```
2657 got_block:
2658
2659   fix_mutex = buf_page_get_mutex(&fix_block->page);
2660
2661   ut_ad(page_zip_get_size(&block->page.zip) == zip_size);
2662
2663   if (mode == BUF_GET_IF_IN_POOL || mode == BUF_PEEK_IF_IN_POOL) {
2664
2665     bool  must_read;
2666
2667     {
2668       buf_page_t* fix_page = &fix_block->page;
2669
2670       mutex_enter(fix_mutex);
2671
2672       buf_io_fix  io_fix = buf_page_get_io_fix(fix_page);
2673
2674       must_read = (io_fix == BUF_IO_READ);
2675
2676       mutex_exit(fix_mutex);
2677     }
2678
2679     if (must_read) {
2680       /* The page is being read to buffer pool,
2681       but we cannot wait around for the read to
2682       complete. */
2683       buf_block_unfix(fix_block);
2684
2685       return(NULL);
2686     }
2687   }
```

mode is buffer get if in pool

Still reading

Buffer Read - get page mode

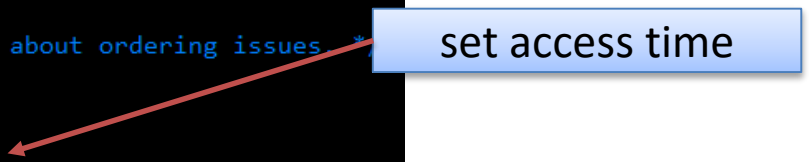
- include/buf0buf.h

```
41 /** @name Modes for buf_page_get_gen */
42 /* @{ */
43 #define BUF_GET     10  /*!< get always */
44 #define BUF_GET_IF_IN_POOL  11  /*!< get if in pool */
45 #define BUF_PEEK_IF_IN_POOL 12  /*!< get if in pool, do not make
46           the block young in the LRU list */
47 #define BUF_GET_NO_LATCH  14  /*!< get and bufferfix, but
48           set no latch; we have
49           separated this case, because
50           it is error-prone programming
51           not to set a latch, and it
52           should be used with care */
53 #define BUF_GET_IF_IN_POOL_OR_WATCH 15
54           /*!< Get the page only if it's in the
55           buffer pool, if not then set a watch
56           on the page. */
57 #define BUF_GET_POSSIBLY_FREED    16
58           /*!< Like BUF_GET, but do not mind
59           if the file page has been freed. */
```

Buffer Read

- buf/buf0buf.cc:2491, buf_page_get_gen()

```
2955   ut_ad(fix_block->page.buf_fix_count > 0);
2956
2957 #ifdef UNIV_SYNC_DEBUG
2958   /* We have already buffer fixed the page, and we are committed to
2959   returning this page to the caller. Register for debugging. */
2960   {
2961     ibool ret;
2962     ret = rw_lock_s_lock_nowait(&fix_block->debug_latch, file, line);
2963     ut_a(ret);
2964   }
2965 #endif /* UNIV_SYNC_DEBUG */
2966
2967 #if defined UNIV_DEBUG_FILE_ACCESSES || defined UNIV_DEBUG
2968   ut_a(mode == BUF_GET_POSSIBLY_FREED
2969        || !fix_block->page.file_page_was_freed);
2970 #endif
2971   /* Check if this is the first access to the page */
2972   access_time = buf_page_is_accessed(&fix_block->page);
2973
2974   /* This is a heuristic and we don't care about ordering issues. */
2975   if (access_time == 0) {
2976     buf_block_mutex_enter(fix_block);
2977
2978     buf_page_set_accessed(&fix_block->page);
2979
2980     buf_block_mutex_exit(fix_block);
2981   }
2982
2983   if (mode != BUF_PEEK_IF_IN_POOL) {
2984     buf_page_make_young_if_needed(&fix_block->page);
2985   }
```



Buffer Read

- buf/buf0buf.cc:2491, buf_page_get_gen()

```
2993 #ifdef PAGE_ATOMIC_REF_COUNT
2994   /* We have to wait here because the IO_READ state was set
2995   under the protection of the hash_lock and the block->mutex
2996   but not the block->lock. */
2997   buf_wait_for_read(fix_block);
2998 #endif /* PAGE_ATOMIC_REF_COUNT */
2999
3000   switch (rw_latch) {
3001   case RW_NO_LATCH:
3002
3003 #ifndef PAGE_ATOMIC_REF_COUNT
3004     buf_wait_for_read(fix_block);
3005 #endif /* !PAGE_ATOMIC_REF_COUNT */
3006
3007     fix_type = MTR_MEMO_BUF_FIX;
3008     break;
3009
3010   case RW_S_LATCH:
3011     rw_lock_s_lock_inline(&fix_block->lock, 0, file, line);
3012
3013     fix_type = MTR_MEMO_PAGE_S_FIX;
3014     break;
3015
3016   default:
3017     ut_ad(rw_latch == RW_X_LATCH);
3018     rw_lock_x_lock_inline(&fix_block->lock, 0, file, line);
3019
3020     fix_type = MTR_MEMO_PAGE_X_FIX;
3021     break;
3022   }
```

wait until io_fix changed to none

Buffer Read

- buf/buf0buf.cc:2491, buf_page_get_gen()

```
3024   mtr_memo_push(mtr, fix_block, fix_type);
3025
3026   if (mode != BUF_PEEK_IF_IN_POOL && !access_time) {
3027     /* In the case of a first access, try to apply linear
3028     read-ahead */
3029
3030     buf_read_ahead_linear(
3031       space, zip_size, offset, ibuf_inside(mtr));
3032   }
3033
3034 #ifdef UNIV_IBUF_COUNT_DEBUG
3035   ut_a(ibuf_count_get(buf_block_get_space(fix_block),
3036           buf_block_get_page_no(fix_block)) == 0);
3037 #endif
3038 #ifdef UNIV_SYNC_DEBUG
3039   ut_ad(!rw_lock_own(hash_lock, RW_LOCK_EX));
3040   ut_ad(!rw_lock_own(hash_lock, RW_LOCK_SHARED));
3041 #endif /* UNIV_SYNC_DEBUG */
3042   return(fix_block);
3043 }
```

leave mini transaction memo (kind of log)

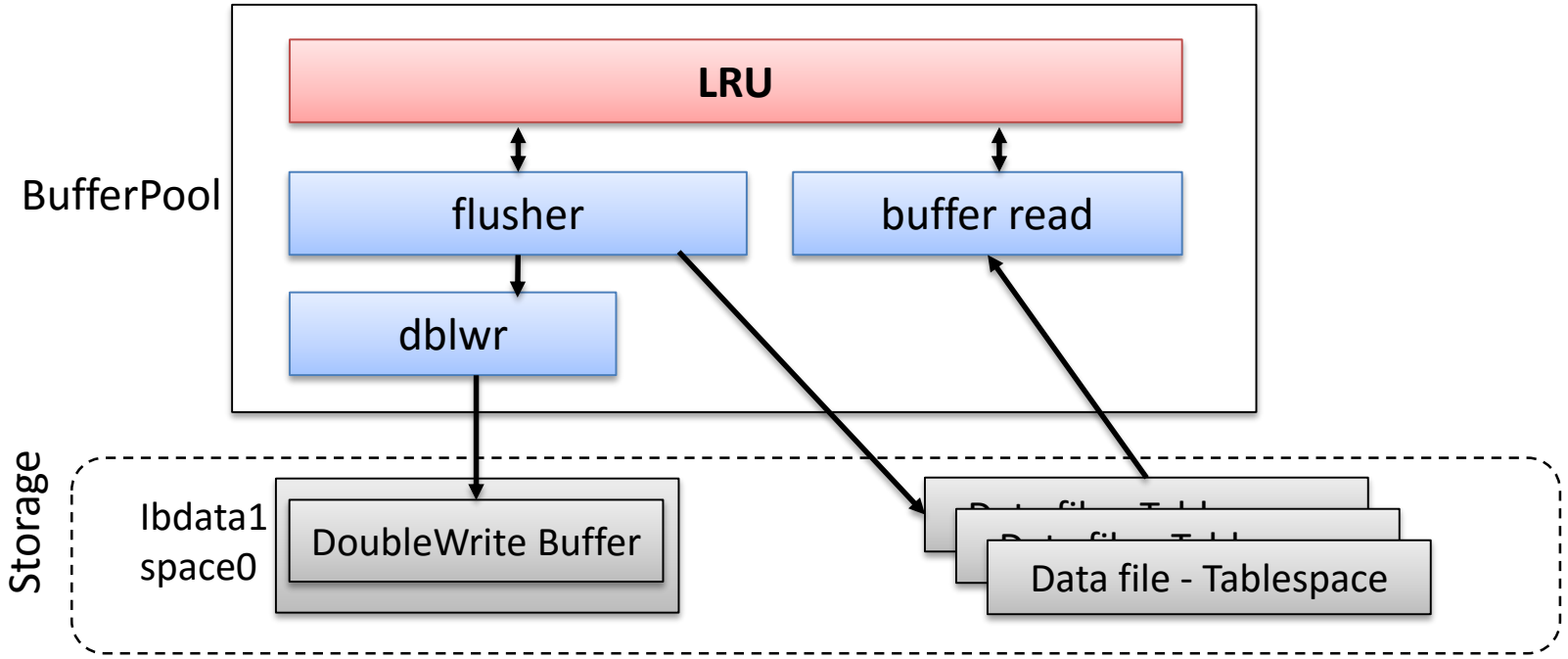
do read ahead (default:false)

Contents

- Buffer manager
  - Overview
  - Buffer Pool (buf0buf.cc)
  - Buffer Read (buf0read.cc)
  - LRU (buf0lru.cc)
  - Flusher (buf0flu.cc)
  - Doublewrite (buf0dblwr.cc)
- File manager
  - Overview
  - file system (fil0fil.cc)
  - OS specific impl. for file system (os0file.cc)

Overview

- Buffer Manager
  - Buffer Pool (buf0buf.cc) : Buffer Pool manager
  - Buffer Read (buf0read.cc) : Read Buffer
  - **LRU (buf0lru.cc) : Buffer Replacement**
  - Flusher (buf0flu.cc) : dirty page writer, background flusher
  - Doublewrite (buf0dblwr.cc) : Doublewrite



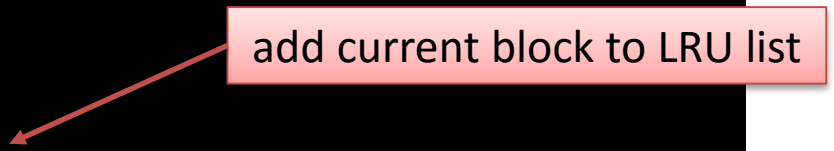
add block

LRU REPLACEMENT

LRU add block

- buf/buf0buf.cc:3477, buf_page_init_for_read()

```
3561    if (block) {
3562      bpage = &block->page;
3563
3564      mutex_enter(&block->mutex);
3565
3566      ut_ad(buf_pool_from_bpage(bpage) == buf_pool);
3567
3568      buf_page_init(buf_pool, space, offset, fold, zip_size, block);
3569
3570 #ifdef PAGE_ATOMIC_REF_COUNT
3571      /* Note: We set the io state without the protection of
3572      the block->lock. This is because other threads cannot
3573      access this block unless it is in the hash table. */
3574
3575      buf_page_set_io_fix(bpage, BUF_IO_READ);
3576 #endif /* PAGE_ATOMIC_REF_COUNT */
3577
3578      rw_lock_x_unlock(hash_lock);
3579
3580      /* The block must be put to the LRU list, to the old blocks */
3581      buf_LRU_add_block(bpage, TRUE/* to old blocks */);
```



LRU add block

- buf/buf0lru.cc:1743, buf_LRU_add_block()

```
1736 /******************************************************************//**
1737 Adds a block to the LRU list. Please make sure that the zip_size is
1738 already set into the page zip when invoking the function, so that we
1739 can get correct zip_size from the buffer page when adding a block
1740 into LRU */
1741 UNIV_INTERN
1742 void
1743 buf_LRU_add_block(
1744 /*==============*/
1745   buf_page_t* bpage,  /*!< in: control block */
1746   ibool   old)  /*!< in: TRUE if should be put to the old
1747         blocks in the LRU list, else put to the start;
1748         if the LRU list is very short, the block is
1749         added to the start, regardless of this
1750         parameter */
1751 {
1752   buf_LRU_add_block_low(bpage, old);
1753 }
```

LRU add block

- buf/buf0lru.cc:1671, buf_LRU_add_block_low()

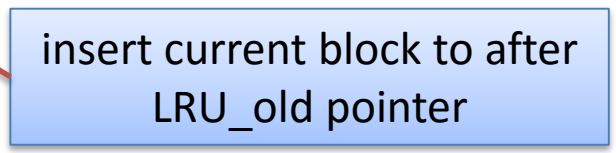
```
1669 UNIV_INLINE
1670 void
1671 buf_LRU_add_block_low(
1672 /*==================*/
1673   buf_page_t* bpage,  /*!< in: control block */
1674   ibool   old)  /*!< in: TRUE if should be put to the old blocks
1675         in the LRU list, else put to the start; if the
1676         LRU list is very short, the block is added to
1677         the start, regardless of this parameter */
1678 {
1679   buf_pool_t* buf_pool = buf_pool_from_bpage(bpage);
1680
1681   ut_ad(buf_pool_mutex_own(buf_pool));
1682
1683   ut_a(buf_page_in_file(bpage));
1684   ut_ad(!bpage->in_LRU_list);
1685
1686   if (!old || (UT_LIST_GET_LEN(buf_pool->LRU) < BUF_LRU_OLD_MIN_LEN)) {
1687
1688     UT_LIST_ADD_FIRST(LRU, buf_pool->LRU, bpage);
1689
1690     bpage->freed_page_clock = buf_pool->freed_page_clock;
1691   } else {
```

check condition : add to first?
if list is too small then put current block to first

LRU add block

- buf/buf0lru.cc:1671, buf_LRU_add_block_low()

```
1691    } else {
1692 #ifdef UNIV_LRU_DEBUG
1693      /* buf_pool->LRU_old must be the first item in the LRU list
1694      whose "old" flag is set. */
1695      ut_a(buf_pool->LRU_old->old);
1696      ut_a(!UT_LIST_GET_PREV(LRU, buf_pool->LRU_old)
1697           || !UT_LIST_GET_PREV(LRU, buf_pool->LRU_old)->old);
1698      ut_a(!UT_LIST_GET_NEXT(LRU, buf_pool->LRU_old)
1699           || UT_LIST_GET_NEXT(LRU, buf_pool->LRU_old)->old);
1700 #endif /* UNIV_LRU_DEBUG */
1701      UT_LIST_INSERT_AFTER(LRU, buf_pool->LRU, buf_pool->LRU_old,
1702                           bpage);
1703      buf_pool->LRU_old_len++;
1704    }
```



LRU add block

- buf/buf0lru.cc:1671, buf_LRU_add_block_low()

```
1706   ut_d(bpage->in_LRU_list = TRUE);
1707
1708   incr_LRU_size_in_bytes(bpage, buf_pool);
1709
1710   if (UT_LIST_GET_LEN(buf_pool->LRU) > BUF_LRU_OLD_MIN_LEN) {
1711
1712     ut_ad(buf_pool->LRU_old);
1713
1714     /* Adjust the length of the old block list if necessary */
1715
1716     buf_page_set_old(bpage, old);
1717     buf_LRU_old_adjust_len(buf_pool);
```

if current buffer pool length is larger than old_min

```
564 UNIV_INLINE
565 void
566 buf_page_set_old(
567 /*=============*/
568   buf_page_t* bpage,  /*!< in/out: control block */
569   ibool   old)  /*!< in: old */
570 {
571   ut_a(buf_page_in_file(bpage));
572   ut_ad(buf_pool_mutex_own(buf_pool));
573   ut_ad(bpage->in_LRU_list);
574
575   bpage->old = old;
576 }
```

LRU add block

- buf/buf0lru.cc:1671, buf_LRU_add_block_low()

```
1706   ut_d(bpage->in_LRU_list = TRUE);
1707
1708   incr_LRU_size_in_bytes(bpage, buf_pool);
1709
1710   if (UT_LIST_GET_LEN(buf_pool->LRU) > BUF_LRU_OLD_MIN_LEN) {
1711
1712     ut_ad(buf_pool->LRU_old);
1713
1714     /* Adjust the length of the old block list if necessary */
1715
1716     buf_page_set_old(bpage, old);
1717     buf_LRU_old_adjust_len(buf_pool);
1718
1719   } else if (UT_LIST_GET_LEN(buf_pool->LRU) == BUF_LRU_OLD_MIN_LEN) {
1720
1721     /* The LRU list is now long enough for LRU_old to become
1722     defined: init it */
1723
1724     buf_LRU_old_init(buf_pool);
1725   } else {
1726     buf_page_set_old(bpage, buf_pool->LRU_old != NULL);
1727   }
1728
1729   /* If this is a zipped block with decompressed frame as well
1730   then put it on the unzip_LRU list */
1731   if (buf_page_belongs_to_unzip_LRU(bpage)) {
1732     buf_unzip_LRU_add_block((buf_block_t*) bpage, old);
1733   }
1734 }
```

if current buffer pool length is larger than old_min

Adjust length old blocks

LRU add block - old_adjust_len

- buf/buf0lru.cc:1384, buf_LRU_old_adjust_len()

```
1382 UNIV_INLINE
1383 void
1384 buf_LRU_old_adjust_len(
1385 /*===================*/
1386   buf_pool_t* buf_pool) /*!< in: buffer pool instance */
1387 {
1388   ulint old_len;
1389   ulint new_len;
1390
1391   ut_a(buf_pool->LRU_old);
1392   ut_ad(buf_pool_mutex_own(buf_pool));
1393   ut_ad(buf_pool->LRU_old_ratio >= BUF_LRU_OLD_RATIO_MIN);
1394   ut_ad(buf_pool->LRU_old_ratio <= BUF_LRU_OLD_RATIO_MAX);
1395 #if BUF_LRU_OLD_RATIO_MIN * BUF_LRU_OLD_MIN_LEN <= BUF_LRU_OLD_RATIO_DIV * (BUF_LRU_OLD_TOLERANCE + 5)
1396 # error "BUF_LRU_OLD_RATIO_MIN * BUF_LRU_OLD_MIN_LEN <= BUF_LRU_OLD_RATIO_DIV * (BUF_LRU_OLD_TOLERANCE + 5)"
1397 #endif
1398 #ifdef UNIV_LRU_DEBUG
1399   /* buf_pool->LRU_old must be the first item in the LRU list
1400   whose "old" flag is set. */
1401   ut_a(buf_pool->LRU_old->old);
1402   ut_a(!UT_LIST_GET_PREV(LRU, buf_pool->LRU_old)
1403        || !UT_LIST_GET_PREV(LRU, buf_pool->LRU_old)->old);
1404   ut_a(!UT_LIST_GET_NEXT(LRU, buf_pool->LRU_old)
1405        || UT_LIST_GET_NEXT(LRU, buf_pool->LRU_old)->old);
1406 #endif /* UNIV_LRU_DEBUG */
1407
```

LRU add block - old_adjust_len

- buf/buf0lru.cc:1384, buf_LRU_old_adjust_len()

```
1408    old_len = buf_pool->LRU_old_len;
1409    new_len = ut_min(UT_LIST_GET_LEN(buf_pool->LRU)
1410         * buf_pool->LRU_old_ratio / BUF_LRU_OLD_RATIO_DIV,
1411         UT_LIST_GET_LEN(buf_pool->LRU)
1412         - (BUF_LRU_OLD_TOLERANCE
1413            + BUF_LRU_NON_OLD_MIN_LEN));
```

adjust old LRU length based on heuristics

BUF_LRU_OLD_RATIO_DIV : 1024
BUF_LRU_OLD_TORRANCE : 20
BUF_LRU_NON_OLD_MIN_LEN : 5

if the buffer is large enough, then old buffer length become **(buffer_len*3/8)/1024**

LRU add block - old_adjust_len

- buf/buf0lru.cc:1384, buf_LRU_old_adjust_len()

```
1415   for (;;) {
1416     buf_page_t* LRU_old = buf_pool->LRU_old;
1417
1418     ut_a(LRU_old);
1419     ut_ad(LRU_old->in_LRU_list);
1420 #ifdef UNIV_LRU_DEBUG
1421     ut_a(LRU_old->old);
1422 #endif /* UNIV_LRU_DEBUG */
1423
1424     /* Update the LRU_old pointer if necessary */
1425
1426     if (old_len + BUF_LRU_OLD_TOLERANCE < new_len) {
1427
1428       buf_pool->LRU_old = LRU_old = UT_LIST_GET_PREV(
1429         LRU, LRU_old);
1430 #ifdef UNIV_LRU_DEBUG
1431       ut_a(!LRU_old->old);
1432 #endif /* UNIV_LRU_DEBUG */
1433       old_len = ++buf_pool->LRU_old_len;
1434       buf_page_set_old(LRU_old, TRUE);
1435
1436     } else if (old_len > new_len + BUF_LRU_OLD_TOLERANCE) {
1437
1438       buf_pool->LRU_old = UT_LIST_GET_NEXT(LRU, LRU_old);
1439       old_len = --buf_pool->LRU_old_len;
1440       buf_page_set_old(LRU_old, FALSE);
1441     } else {
1442       return;
1443     }
1444   }
1445 }
```

Update LRU_old pointer

Increase old blocks

decrease old blocks

LRU add block

- buf/buf0lru.cc:1671, buf_LRU_add_block_low()

```
1706   ut_d(bpage->in_LRU_list = TRUE);
1707
1708   incr_LRU_size_in_bytes(bpage, buf_pool);
1709
1710   if (UT_LIST_GET_LEN(buf_pool->LRU) > BUF_LRU_OLD_MIN_LEN) {
1711
1712     ut_ad(buf_pool->LRU_old);
1713
1714     /* Adjust the length of the old block list if necessary */
1715
1716     buf_page_set_old(bpage, old);
1717     buf_LRU_old_adjust_len(buf_pool);
1718
1719   } else if (UT_LIST_GET_LEN(buf_pool->LRU) == BUF_LRU_OLD_MIN_LEN) {
1720
1721     /* The LRU list is now long enough for LRU_old to become
1722     defined: init it */
1723
1724     buf_LRU_old_init(buf_pool);
1725   } else {
1726     buf_page_set_old(bpage, buf_pool->LRU_old != NULL);
1727   }
1728
1729   /* If this is a zipped block with decompressed frame as well
1730   then put it on the unzip_LRU list */
1731   if (buf_page_belongs_to_unzip_LRU(bpage)) {
1732     buf_unzip_LRU_add_block((buf_block_t*) bpage, old);
1733   }
1734 }
```

if current buffer pool length is larger than old_min

init old pointer

if current buffer is small, then just set this old

LRU add block - old_init

- buf/buf0lru.cc:1452, buf_LRU_old_init()

```
1450 static
1451 void
1452 buf_LRU_old_init(
1453 /*=============*/
1454   buf_pool_t* buf_pool)
1455 {
1456   buf_page_t* bpage;
1457
1458   ut_ad(buf_pool_mutex_own(buf_pool));
1459   ut_a(UT_LIST_GET_LEN(buf_pool->LRU) == BUF_LRU_OLD_MIN_LEN);
1460
1461   /* We first initialize all blocks in the LRU list as old and then use
1462   the adjust function to move the LRU_old pointer to the right
1463   position */
1464
1465   for (bpage = UT_LIST_GET_LAST(buf_pool->LRU); bpage != NULL;
1466        bpage = UT_LIST_GET_PREV(LRU, bpage)) {
1467     ut_ad(bpage->in_LRU_list);
1468     ut_ad(buf_page_in_file(bpage));
1469     /* This loop temporarily violates the
1470     assertions of buf_page_set_old(). */
1471     bpage->old = TRUE;
1472   }
1473
1474   buf_pool->LRU_old = UT_LIST_GET_FIRST(buf_pool->LRU);
1475   buf_pool->LRU_old_len = UT_LIST_GET_LEN(buf_pool->LRU);
1476
1477   buf_LRU_old_adjust_len(buf_pool);
1478 }
```

Mark every blocks as old

adjust old length

get free block

LRU REPLACEMENT

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.**

- 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
    - the above will put the block on free list
    - success: retry the free list

- iteration 1:
  - same as iteration 0 except:
    - scan whole LRU list
    - scan LRU list even if buf_pool->try_LRU_scan is not set

- iteration > 1:
  - same as iteration 1 but sleep 100ms

LRU get free block

- buf/buf0lru.cc:1238, buf_LRU_get_free_block()

```
1236 UNIV_INTERN
1237 buf_block_t*
1238 buf_LRU_get_free_block(
1239 /*=====================*/
1240   buf_pool_t* buf_pool) /*!< in/out: buffer pool instance */
1241 {
1242   buf_block_t*  block   = NULL;
1243   ibool   freed   = FALSE;
1244   ulint   n_iterations  = 0;
1245   ulint   flush_failures  = 0;
1246   ibool   mon_value_was = FALSE;
1247   ibool   started_monitor = FALSE;
1248
1249   MONITOR_INC(MONITOR_LRU_GET_FREE_SEARCH);
1250 loop:
1251   buf_pool_mutex_enter(buf_pool);
1252
1253   buf_LRU_check_size_of_non_data_objects(buf_pool);
1254
1255   /* If there is a block in the free list, take it */
1256   block = buf_LRU_get_free_only(buf_pool);
1257
1258   if (block) {
1259
1260     buf_pool_mutex_exit(buf_pool);
1261     ut_ad(buf_pool_from_block(block) == buf_pool);
1262     memset(&block->page.zip, 0, sizeof block->page.zip);
1263
1264     if (started_monitor) {
1265       srv_print_innodb_monitor =
1266         static_cast<my_bool>(mon_value_was);
1267     }
1268
1269     return(block);
1270   }
```

get free block from the free list

LRU get free block - get free block from fre list

- `buf/buf0lru.cc:1100, buf_LRU_get_free_only()`

```
1098 UNIV_INTERN
1099 buf_block_t*
1100 buf_LRU_get_free_only(
1101 /*==================*/
1102   buf_pool_t* buf_pool)
1103 {
1104   buf_block_t*  block;
1105
1106   ut_ad(buf_pool_mutex_own(buf_pool));
1107
1108   block = (buf_block_t*) UT_LIST_GET_FIRST(buf_pool->free);
1109
1110   if (block) {
1111
1112     ut_ad(block->page.in_free_list);
1113     ut_d(block->page.in_free_list = FALSE);
1114     ut_ad(!block->page.in_flush_list);
1115     ut_ad(!block->page.in_LRU_list);
1116     ut_a(!buf_page_in_file(&block->page));
1117     UT_LIST_REMOVE(list, buf_pool->free, (&block->page));
1118
1119     mutex_enter(&block->mutex);
1120
1121     buf_block_set_state(block, BUF_BLOCK_READY_FOR_USE);
1122     UNIV_MEM_ALLOC(block->frame, UNIV_PAGE_SIZE);
1123
1124     ut_ad(buf_pool_from_block(block) == buf_pool);
1125
1126     mutex_exit(&block->mutex);
1127   }
1128
1129   return(block);
1130 }
```

get a block from the free list

if we get a block from the free list, then check state

remove from the list

ignore it : this is used for valgrind

LRU get free block

- buf/buf0lru.cc:1238, buf_LRU_get_free_block()

```
1236 UNIV_INTERN
1237 buf_block_t*
1238 buf_LRU_get_free_block(
1239 /*=====================*/
1240   buf_pool_t* buf_pool) /*!< in/out: buffer pool instance */
1241 {
1242   buf_block_t*  block   = NULL;
1243   ibool   freed   = FALSE;
1244   ulint   n_iterations  = 0;
1245   ulint   flush_failures  = 0;
1246   ibool   mon_value_was = FALSE;
1247   ibool   started_monitor = FALSE;
1248
1249   MONITOR_INC(MONITOR_LRU_GET_FREE_SEARCH);
1250 loop:
1251   buf_pool_mutex_enter(buf_pool);
1252
1253   buf_LRU_check_size_of_non_data_objects(buf_pool);
1254
1255   /* If there is a block in the free list, take it */
1256   block = buf_LRU_get_free_only(buf_pool);
1257
1258   if (block) {
1259
1260     buf_pool_mutex_exit(buf_pool);
1261     ut_ad(buf_pool_from_block(block) == buf_pool);
1262     memset(&block->page.zip, 0, sizeof block->page.zip);
1263
1264     if (started_monitor) {
1265       srv_print_innodb_monitor =
1266         static_cast<my_bool>(mon_value_was);
1267     }
1268
1269     return(block);
1270   }
```

getting a free block succeed

LRU get free block

- `buf/buf0lru.cc:1238, buf_LRU_get_free_block()`

after this line, we need to find a free block, buffer replacement happen

```
1271
1272    if (buf_pool->init_flush[BUF_FLUSH_LRU]
1273        && srv_use_doublewrite_buf
1274        && buf_dblwr != NULL) {
1275
1276      /* If there is an LRU flush happening in the background
1277      then we wait for it to end instead of trying a single
1278      page flush. If, however, we are not using doublewrite
1279      buffer then it is better to do our own single page
1280      flush instead of waiting for LRU flush to end. */
1281      buf_pool_mutex_exit(buf_pool);
1282      buf_flush_wait_batch_end(buf_pool, BUF_FLUSH_LRU);
1283      goto loop;
1284    }
```

already background flusher started, see this later

*note : init_flush[BUF_FLUSH_LRU]=true
after background flusher started for current buffer pool instance
see this later

LRU get free block

- buf/buf0lru.cc:1238, buf_LRU_get_free_block()

```
1286   freed = FALSE;
1287   if (buf_pool->try_LRU_scan || n_iterations > 0) {
1288     /* If no block was in the free list, search from the
1289     end of the LRU list and try to free a block there.
1290     If we are doing for the first time we'll scan only
1291     tail of the LRU list otherwise we scan the whole LRU
1292     list. */
1293     freed = buf_LRU_scan_and_free_block(buf_pool,
1294                 n_iterations > 0);
1295
1296     if (!freed && n_iterations == 0) {
1297       /* Tell other threads that there is no point
1298       in scanning the LRU list. This flag is set to
1299       TRUE again when we flush a batch from this
1300       buffer pool. */
1301       buf_pool->try_LRU_scan = FALSE;
1302     }
1303   }
1304
1305   buf_pool_mutex_exit(buf_pool);
1306
1307   if (freed) {
1308     goto loop;
1309
1310   }
```

find a victim buffer to replace and make a free buffer

this means we have free buffer(s) -> goto loop

freed : # of free buffer just made

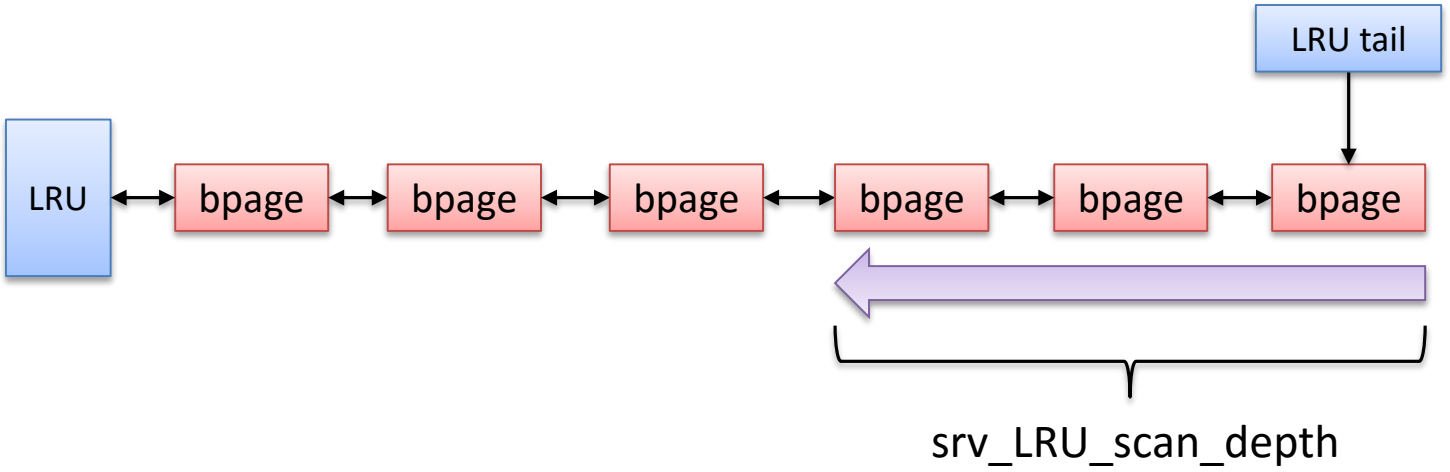
LRU get free block - scan and free block

- buf/buf0lru.cc:1046, buf_LRU_scan_and_free_block()

```
1041 /******************************************************************//**
1042 Try to free a replaceable block.
1043 @return TRUE if found and freed */
1044 UNIV_INTERN
1045 ibool
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 - scan and free block

- `buf/buf0lru.cc:1046, buf_LRU_scan_and_free_block()`



LRU get free block - scan and free block

- `buf/buf0lru.cc:1046, buf_LRU_scan_and_free_block()`

```
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  }
```

get last -> tail

if we find a free block then stop finding

we're going to scan the buffers only the scan_depth amount from the tail

* Try to free it :
if it is dirty, write to database.
Then, add it to free list

LRU get free block - free page

- `buf/buf0lru.cc:1800, buf_LRU_free_page()`
  - get hash lock : x-lock it
  - get block mutex
  - check it can be relocate (replaceable)
    - IO_FIX == IO_NONE
    - BUFFER FIX COUNT == 0
  - free it

LRU get free block - free page

- buf/buf0lru.cc:1800, buf_LRU_free_page()

```
1800 buf_LRU_free_page(
1801 /*===============*/
1802   buf_page_t* bpage,  /*!< in: block to be freed */
1803   bool    zip)  /*!< in: true if should remove also the
1804         compressed page of an uncompressed page */
1805 {
1806   buf_page_t* b = NULL;
1807   buf_pool_t* buf_pool = buf_pool_from_bpage(bpage);
1808   const ulint fold = buf_page_address_fold(bpage->space,
1809                 bpage->offset);
1810   rw_lock_t*  hash_lock = buf_page_hash_lock_get(buf_pool, fold);
1811
1812   ib_mutex_t* block_mutex = buf_page_get_mutex(bpage);
1813
1814   ut_ad(buf_pool_mutex_own(buf_pool));
1815   ut_ad(buf_page_in_file(bpage));
1816   ut_ad(bpage->in_LRU_list);
1817
1818   rw_lock_x_lock(hash_lock);
1819   mutex_enter(block_mutex);
1820
1821   if (!buf_page_can_relocate(bpage)) {
1822
1823     /* Do not free buffer fixed or I/O-fixed blocks. */
1824     goto func_exit;
1825   }
```

get hash lock and block mutex

check can relocate

LRU get free block - free page, relocate

- buf/buf0buf.ic:528, buf_page_can_relocate()

```
523 /*********************************************************************//**
524 Determine if a buffer block can be relocated in memory.  The block
525 can be dirty, but it must not be I/O-fixed or bufferfixed. */
526 UNIV_INLINE
527 ibool
528 buf_page_can_relocate(
529 /*==================*/
530   const buf_page_t* bpage)  /*!< control block being relocated */
531 {
532 #ifdef UNIV_DEBUG
533   buf_pool_t* buf_pool = buf_pool_from_bpage(bpage);
534   ut_ad(buf_pool_mutex_own(buf_pool));
535 #endif
536   ut_ad(mutex_own(buf_page_get_mutex(bpage)));
537   ut_ad(buf_page_in_file(bpage));
538   ut_ad(bpage->in_LRU_list);
539
540   return(buf_page_get_io_fix(bpage) == BUF_IO_NONE
541          && bpage->buf_fix_count == 0);
542 }
```

LRU get free block - free page

- `buf/buf0lru.cc:1800, buf_LRU_free_page()`

```
1831  if (zip || !bpage->zip.data) {
1832    /* This would completely free the block. */
1833    /* Do not completely free dirty blocks. */
1834
1835    if (bpage->oldest_modification) {
1836      goto func_exit;
1837    }
1838  } else if (bpage->oldest_modification > 0
1839       && buf_page_get_state(bpage) != BUF_BLOCK_FILE_PAGE) {
1840
1841    ut_ad(buf_page_get_state(bpage) == BUF_BLOCK_ZIP_DIRTY);
1842
1843 func_exit:
1844    rw_lock_x_unlock(hash_lock);
1845    mutex_exit(block_mutex);
1846    return(false);
1847
1848  } else if (buf_page_get_state(bpage) == BUF_BLOCK_FILE_PAGE) {
1849    b = buf_page_alloc_descriptor();
1850    ut_a(b);
1851    memcpy(b, bpage, sizeof *b);
1852  }
```

if this true, then current buffer is dirty and not flushed to disk yet

after this block, we're clean case and check relocation then remove from the hash table

LRU get free block - free page

- `buf/buf0lru.cc:1800, buf_LRU_free_page()`

clean block case

```
1854   ut_ad(buf_pool_mutex_own(buf_pool));
1855   ut_ad(buf_page_in_file(bpage));
1856   ut_ad(bpage->in_LRU_list);
1857   ut_ad(!bpage->in_flush_list == !bpage->oldest_modification);
1858
1859 #ifdef UNIV_DEBUG
1860   if (buf_debug_prints) {
1861     fprintf(stderr, "Putting space %lu page %lu to free list\n",
1862       (ulong) buf_page_get_space(bpage),
1863       (ulong) buf_page_get_page_no(bpage));
1864   }
1865 #endif /* UNIV_DEBUG */
1866
1867 #ifdef UNIV_SYNC_DEBUG
1868         ut_ad(rw_lock_own(hash_lock, RW_LOCK_EX));
1869 #endif /* UNIV_SYNC_DEBUG */
1870   ut_ad(buf_page_can_relocate(bpage));
1871
1872   if (!buf_LRU_block_remove_hashed(bpage, zip)) {
1873     return(true);
1874   }
```

LRU get free block - free page

- buf/buf0lru.cc:2129, buf_LRU_block_remove_hashed()

```
2129 buf_LRU_block_remove_hashed(
2130 /*=============================*/
2131   buf_page_t* bpage,  /*!< in: block, must contain a file page and
2132         be in a state where it can be freed; there
2133         may or may not be a hash index to the page */
2134   bool    zip)  /*!< in: true if should remove also the
2135         compressed page of an uncompressed page */
2136 {
2137   ulint     fold;
2138   const buf_page_t* hashed_bpage;
2139   buf_pool_t*   buf_pool = buf_pool_from_bpage(bpage);
2140   rw_lock_t*    hash_lock;
2141
2142   ut_ad(bpage);
2143   ut_ad(buf_pool_mutex_own(buf_pool));
2144   ut_ad(mutex_own(buf_page_get_mutex(bpage)));
2145
2146   fold = buf_page_address_fold(bpage->space, bpage->offset);
2147   hash_lock = buf_page_hash_lock_get(buf_pool, fold);
2148 #ifdef UNIV_SYNC_DEBUG
2149         ut_ad(rw_lock_own(hash_lock, RW_LOCK_EX));
2150 #endif /* UNIV_SYNC_DEBUG */
2151
2152   ut_a(buf_page_get_io_fix(bpage) == BUF_IO_NONE);
2153   ut_a(bpage->buf_fix_count == 0);
2154
2155   buf_LRU_remove_block(bpage);
2156
2157   buf_pool->freed_page_clock += 1;
```

remove block from the LRU list

LRU get free block - free page

- `buf/buf0lru.cc:2129, buf_LRU_block_remove_hashed()`

```
2230   hashed_bpage = buf_page_hash_get_low(buf_pool, bpage->space,
2231                 bpage->offset, fold);
2232
2233   if (UNIV_UNLIKELY(bpage != hashed_bpage)) {
2234     fprintf(stderr,
2235       "InnoDB: Error: page %lu %lu not found"
2236       " in the hash table\n",
2237       (ulong) bpage->space,
2238       (ulong) bpage->offset);
2239     if (hashed_bpage) {
2240       fprintf(stderr,
2241         "InnoDB: In hash table we find block"
2242         " %p of %lu %lu which is not %p\n",
2243         (const void*) hashed_bpage,
2244         (ulong) hashed_bpage->space,
2245         (ulong) hashed_bpage->offset,
2246         (const void*) bpage);
2247     }
2248
2249 #if defined UNIV_DEBUG || defined UNIV_BUF_DEBUG
2250     mutex_exit(buf_page_get_mutex(bpage));
2251     rw_lock_x_unlock(hash_lock);
2252     buf_pool_mutex_exit(buf_pool);
2253     buf_print();
2254     buf_LRU_print();
2255     buf_validate();
2256     buf_LRU_validate();
2257 #endif /* UNIV_DEBUG || UNIV_BUF_DEBUG */
2258     ut_error;
2259   }
```

error handling

LRU get free block - free page

- buf/buf0lru.cc:2129, buf_LRU_block_remove_hashed()

```
2261   ut_ad(!bpage->in_zip_hash);
2262   ut_ad(bpage->in_page_hash);
2263   ut_d(bpage->in_page_hash = FALSE);
2264   HASH_DELETE(buf_page_t, hash, buf_pool->page_hash, fold, bpage);
2265   switch (buf_page_get_state(bpage)) {
2289   case BUF_BLOCK_FILE_PAGE:
2290     memset(((buf_block_t*) bpage)->frame
2291            + FIL_PAGE_OFFSET, 0xff, 4);
2292     memset(((buf_block_t*) bpage)->frame
2293            + FIL_PAGE_ARCH_LOG_NO_OR_SPACE_ID, 0xff, 4);
2294     UNIV_MEM_INVALID(((buf_block_t*) bpage)->frame,
2295          UNIV_PAGE_SIZE);
2296     buf_page_set_state(bpage, BUF_BLOCK_REMOVE_HASH);
2297
2298     if (buf_pool->flush_rbt == NULL) {
2299       bpage->space = ULINT32_UNDEFINED;
2300       bpage->offset = ULINT32_UNDEFINED;
2301     }
2322     rw_lock_x_unlock(hash_lock);
2323     mutex_exit(&((buf_block_t*) bpage)->mutex);
```

HASH_DELETE

LRU get free block

- `buf/buf0lru.cc:1238, buf_LRU_get_free_block()`

back to get free - assume : we couldn't make a free block
- do a single page flush

```
1358   /* No free block was found: try to flush the LRU list.
1359   This call will flush one page from the LRU and put it on the
1360   free list. That means that the free block is up for grabs for
1361   all user threads.
1362   TODO: A more elegant way would have been to return the freed
1363   up block to the caller here but the code that deals with
1364   removing the block from page_hash and LRU_list is fairly
1365   involved (particularly in case of compressed pages). We
1366   can do that in a separate patch sometime in future. */
1367   if (!buf_flush_single_page_from_LRU(buf_pool)) {
1368     MONITOR_INC(MONITOR_LRU_SINGLE_FLUSH_FAILURE_COUNT);
1369     ++flush_failures;
1370   }
1371
1372   srv_stats.buf_pool_wait_free.add(n_iterations, 1);
1373
1374   n_iterations++;
1375
1376   goto loop;
1377 }
```

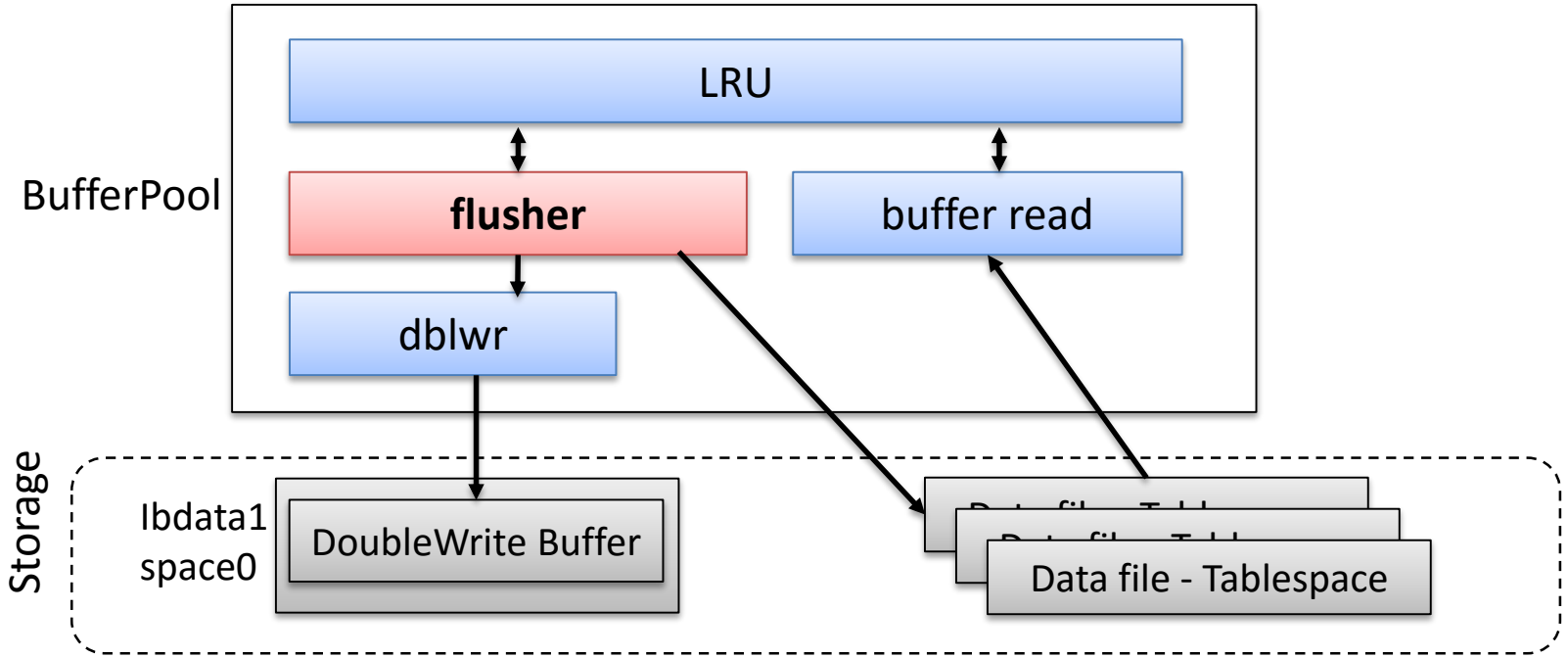
single page flush

part 1: single page flush

FLUSH A PAGE

Overview

- Buffer Manager
  - Buffer Pool (buf0buf.cc) : Buffer Pool manager
  - Buffer Read (buf0read.cc) : Read Buffer
  - LRU (buf0lru.cc) : Buffer Replacement
  - **Flusher (buf0flu.cc) : dirty page writer, background flusher**
  - Doublewrite (buf0dblwr.cc) : Doublewrite

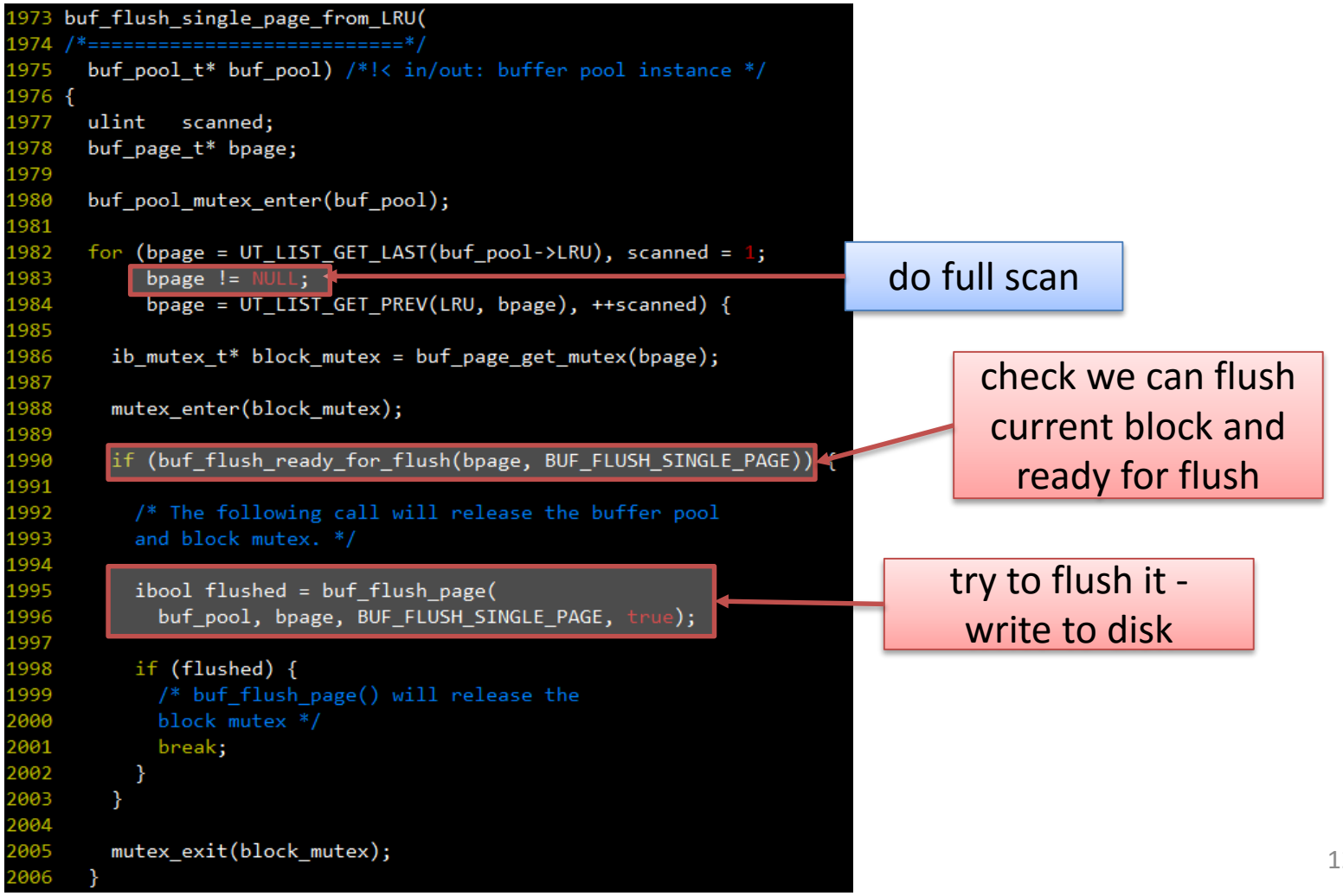


Flush a page

- Flushing page by
  - a background flusher - batch_LRU
  - a single point flush (in LRU_get_free_block)
- Background flusher
  - Regularly check system status
  - flush all buffer pool instance in a batch manner

Single page flush

- buf/buf0flu.cc:1973, buf_flush_single_page_from_LRU()



Single page flush - ready for flush

- buf/buf0flu.cc:532, buf_flush_ready_for_flush()

```
531 bool
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 {
538 #ifdef UNIV_DEBUG
539   buf_pool_t* buf_pool = buf_pool_from_bpage(bpage);
540   ut_ad(buf_pool_mutex_own(buf_pool));
541 #endif /* UNIV_DEBUG */
542
543   ut_a(buf_page_in_file(bpage));
544   ut_ad(mutex_own(buf_page_get_mutex(bpage)));
545   ut_ad(flush_type < BUF_FLUSH_N_TYPES);
546
547   if (bpage->oldest_modification == 0
548       || buf_page_get_io_fix(bpage) != BUF_IO_NONE) {
549     return(false);
550   }
551
552   ut_ad(bpage->in_flush_list);
553
554   switch (flush_type) {
555   case BUF_FLUSH_LIST:
556   case BUF_FLUSH_LRU:
557   case BUF_FLUSH_SINGLE_PAGE:
558     return(true);
559
560   case BUF_FLUSH_N_TYPES:
561     break;
562   }
563
564   ut_error;
565   return(false);
566 }
```

already flushed or doing IO - return false;

return true;

Single page flush - flush page

- `buf/buf0flu.cc:993, buf_flush_page()`
- Writes a flushable page asynchronously from the buffer pool to a file
  - set io_fix : BUF_IO_WRITE
  - set flush event type
  - write block

Single page flush - flush page

- buf/buf0flu.cc:993, buf_flush_page()

```
 992 bool
 993 buf_flush_page(
 994 /*===========*/
 995   buf_pool_t* buf_pool, /*!< in: buffer pool instance */
 996   buf_page_t* bpage,    /*!< in: buffer control block */
 997   buf_flush_t flush_type, /*!< in: type of flush */
 998   bool    sync)   /*!< in: true if sync IO request */
 999 {
1000   ut_ad(flush_type < BUF_FLUSH_N_TYPES);
1001   ut_ad(buf_pool_mutex_own(buf_pool));
1002   ut_ad(buf_page_in_file(bpage));
1003   ut_ad(!sync || flush_type == BUF_FLUSH_SINGLE_PAGE);
1004
1005   ib_mutex_t* block_mutex = buf_page_get_mutex(bpage);
1006
1007   ut_ad(mutex_own(block_mutex));
1008
1009   ut_ad(buf_flush_ready_for_flush(bpage, flush_type));
1010
1011   bool            is_uncompressed;
1012
1013   is_uncompressed = (buf_page_get_state(bpage) == BUF_BLOCK_FILE_PAGE);
1014   ut_ad(is_uncompressed == (block_mutex != &buf_pool->zip_mutex));
1015
1016   ibool           flush;
1017   rw_lock_t*  rw_lock;
1018   bool            no_fix_count = bpage->buf_fix_count == 0;
1019
```

status check

Single page flush - flush page

- `buf/buf0flu.cc:993, buf_flush_page()`

```
1020  if (!is_uncompressed) {
1021    flush = TRUE;
1022    rw_lock = NULL;
1023
1024  } else if (!(no_fix_count || flush_type == BUF_FLUSH_LIST)) {
1025    /* This is a heuristic, to avoid expensive S attempts. */
1026    flush = FALSE;
1027  } else {
1028
1029    rw_lock = &reinterpret_cast<buf_block_t*>(bpage)->lock;
1030
1031    if (flush_type != BUF_FLUSH_LIST) {
1032      flush = rw_lock_s_lock_gen_nowait(
1033        rw_lock, BUF_IO_WRITE);
1034    } else {
1035      /* Will S lock later */
1036      flush = TRUE;
1037    }
1038  }
```

get FIX LOCK

Single page flush - flush page

- buf/buf0flu.cc:993, buf_flush_page()

```
1040   if (flush) {
1041
1042     /* We are committed to flushing by the time we get here */
1043
1044     buf_page_set_io_fix(bpage, BUF_IO_WRITE);
1045
1046     buf_page_set_flush_type(bpage, flush_type);
1047
1048     if (buf_pool->n_flush[flush_type] == 0) {
1049
1050       os_event_reset(buf_pool->no_flush[flush_type]);
1051     }
1052
1053     ++buf_pool->n_flush[flush_type];
1054
1055     mutex_exit(block_mutex);
1056     buf_pool_mutex_exit(buf_pool);
```

set fix and flush type

Single page flush - flush page

- buf/buf0flu.cc:993, buf_flush_page()

```
1058     if (flush_type == BUF_FLUSH_LIST
1059         && is_uncompressed
1060         && !rw_lock_s_lock_gen_nowait(rw_lock, BUF_IO_WRITE)) {
1061       /* avoiding deadlock possibility involves doublewrite
1062          buffer, should flush it, because it might hold the
1063          another block->lock. */
1064       buf_dblwr_flush_buffered_writes();
1065
1066       rw_lock_s_lock_gen(rw_lock, BUF_IO_WRITE);
1067     }
1068
1069     /* Even though bpage is not protected by any mutex at this
1070        point, it is safe to access bpage, because it is io_fixed and
1071        oldest_modification != 0.  Thus, it cannot be relocated in the
1072        buffer pool or removed from flush_list or LRU_list. */
1073
1074     buf_flush_write_block_low(bpage, flush_type, sync);
1075   }
1076
1077   return(flush);
1078 }
```

Single page flush - flush page

- buf/buf0flu.cc:877, buf_flush_write_block_low()

```
876 void
877 buf_flush_write_block_low(
878 /*=======================*/
879   buf_page_t* bpage,    /*!< in: buffer block to write */
880   buf_flush_t flush_type, /*!< in: type of flush */
881   bool    sync)   /*!< in: true if sync IO request */
882 {
883   ulint zip_size  = buf_page_get_zip_size(bpage);
884   page_t* frame   = NULL;

922   /* Force the log to the disk before writing the modifi
923   log_write_up_to(bpage->newest_modification, LOG_WAIT_ALL_GROUPS, TRUE);
924 #endif
```

flush log (Transaction Log - WAL)

Single page flush - flush page

- buf/buf0flu.cc:877, buf_flush_write_block_low()

```
925   switch (buf_page_get_state(bpage)) {
926   case BUF_BLOCK_POOL_WATCH:
927   case BUF_BLOCK_ZIP_PAGE: /* The page should be dirty. */
928   case BUF_BLOCK_NOT_USED:
929   case BUF_BLOCK_READY_FOR_USE:
930   case BUF_BLOCK_MEMORY:
931   case BUF_BLOCK_REMOVE_HASH:
932     ut_error;
933     break;
934   case BUF_BLOCK_ZIP_DIRTY:
935     frame = bpage->zip.data;
936
937     ut_a(page_zip_verify_checksum(frame, zip_size));
938
939     mach_write_to_8(frame + FIL_PAGE_LSN,
940         bpage->newest_modification);
941     memset(frame + FIL_PAGE_FILE_FLUSH_LSN, 0, 8);
942     break;
943   case BUF_BLOCK_FILE_PAGE:
944     frame = bpage->zip.data;
945     if (!frame) {
946       frame = ((buf_block_t*) bpage)->frame;
947     }
948
949     buf_flush_init_for_writing(((buf_block_t*) bpage)->frame,
950             bpage->zip.data
951             ? &bpage->zip : NULL,
952             bpage->newest_modification);
953     break;
954   }
```

Single page flush - flush page

- buf/buf0flu.cc:877, buf_flush_write_block_low()

```
956   if (!srv_use_doublewrite_buf || !buf_dblwr) {
957     fil_io(OS_FILE_WRITE | OS_AIO_SIMULATED_WAKE_LATER,
958            sync, buf_page_get_space(bpage), zip_size,
959            buf_page_get_page_no(bpage), 0,
960            zip_size ? zip_size : UNIV_PAGE_SIZE,
961            frame, bpage);
962   } else if (flush_type == BUF_FLUSH_SINGLE_PAGE) {
963     buf_dblwr_write_single_page(bpage, sync);
964   } else {
965     ut_ad(!sync);
966     buf_dblwr_add_to_batch(bpage);
967   }
968
969   /* When doing single page flushing the IO is done synchronously
970   and we flush the changes to disk only for the tablespace we
971   are working on. */
972   if (sync) {
973     ut_ad(flush_type == BUF_FLUSH_SINGLE_PAGE);
974     fil_flush(buf_page_get_space(bpage));
975     buf_page_io_complete(bpage);
976   }
977
978   /* Increment the counter of I/O operations used
979   for selecting LRU policy. */
980   buf_LRU_stat_inc_io();
981 }
```

double write off case

do single page double write then write to datafile, **see this in dblwr**

sync buffered write to disk : call fsync by **fil_flush**

Single page flush

- buf/buf0flu.cc:1973, buf_flush_single_page_from_LRU()

```
1973 buf_flush_single_page_from_LRU(
1974 /*==========================*/
1975   buf_pool_t* buf_pool) /*!< in/out: buffer pool instance */
1976 {
1977   ulint   scanned;
1978   buf_page_t* bpage;
1979
1980   buf_pool_mutex_enter(buf_pool);
1981
1982   for (bpage = UT_LIST_GET_LAST(buf_pool->LRU), scanned = 1;
1983        bpage != NULL;
1984        bpage = UT_LIST_GET_PREV(LRU, bpage), ++scanned) {
1985
1986     ib_mutex_t* block_mutex = buf_page_get_mutex(bpage);
1987
1988     mutex_enter(block_mutex);
1989
1990     if (buf_flush_ready_for_flush(bpage, BUF_FLUSH_SINGLE_PAGE)) {
1991
1992       /* The following call will release the buffer pool
1993       and block mutex. */
1994
1995       ibool flushed = buf_flush_page(
1996         buf_pool, bpage, BUF_FLUSH_SINGLE_PAGE, true);
1997
1998       if (flushed) {
1999         /* buf_flush_page() will release the
2000         block mutex */
2001         break;
2002       }
2003     }
2004
2005     mutex_exit(block_mutex);
2006   }
```

we just finished this.

Single page flush

- `buf/buf0flu.cc:1973, buf_flush_single_page_from_LRU()`

```
2008  MONITOR_INC_VALUE_CUMULATIVE(
2009    MONITOR_LRU_SINGLE_FLUSH_SCANNED,
2010    MONITOR_LRU_SINGLE_FLUSH_SCANNED_NUM_CALL,
2011    MONITOR_LRU_SINGLE_FLUSH_SCANNED_PER_CALL,
2012    scanned);
2013
2014  if (bpage == NULL) {
2015    /* Can't find a single flushable page. */
2016    buf_pool_mutex_exit(buf_pool);
2017    return(FALSE);
2018  }
```

single page flush failure case

part 2: batch flush

BATCH FLUSHING

background flusher and batch flushing

- page_cleaner thread
  - independent thread for flushing a dirty pages from buffer pools
  - regularly do flush from LRU_tail or
  - do flush by dirty page percent (configurable)
- Thread definition
  - `buf/buf0flu.cc:2389, DECLARE_THREAD(buf_flush_page_cleaner_thread)`

background flusher

- buf/buf0flu.cc:2389, DECLARE_THREAD(buf_flush_page_cleaner_thread)

```
2387 extern "C" UNIV_INTERN
2388 os_thread_ret_t
2389 DECLARE_THREAD(buf_flush_page_cleaner_thread)(
2390 /*=============================================*/
2391   void* arg __attribute__((unused)))
2392       /*!< in: a dummy parameter required by
2393       os_thread_create */
2394 {
2395   ulint next_loop_time = ut_time_ms() + 1000;
2396   ulint n_flushed = 0;
2397   ulint last_activity = srv_get_activity_count();
2398
2399   ut_ad(!srv_read_only_mode);
2400
2401 #ifdef UNIV_PFS_THREAD
2402   pfs_register_thread(buf_page_cleaner_thread_key);
2403 #endif /* UNIV_PFS_THREAD */
2404
2405 #ifdef UNIV_DEBUG_THREAD_CREATION
2406   fprintf(stderr, "InnoDB: page_cleaner thread running, id %lu\n",
2407     os_thread_pf(os_thread_get_curr_id()));
2408 #endif /* UNIV_DEBUG_THREAD_CREATION */
2409
2410   buf_page_cleaner_is_active = TRUE;
```

background flusher

- buf/buf0flu.cc:2389, DECLARE_THREAD(buf_flush_page_cleaner_thread)

```
2412  while (srv_shutdown_state == SRV_SHUTDOWN_NONE) {
2413
2414    /* The page_cleaner skips sleep if the server is
2415    idle and there are no pending IOs in the buffer pool
2416    and there is work to do. */
2417    if (srv_check_activity(last_activity)
2418        || buf_get_n_pending_read_ios()
2419        || n_flushed == 0) {
2420      page_cleaner_sleep_if_needed(next_loop_time);
2421    }
2422
2423    next_loop_time = ut_time_ms() + 1000;
2424
2425    if (srv_check_activity(last_activity)) {
2426      last_activity = srv_get_activity_count();
2427
2428      /* Flush pages from end of LRU if required */
2429      n_flushed = buf_flush_LRU_tail();
2430
2431      /* Flush pages from flush_list if required */
2432      n_flushed += page_cleaner_flush_pages_if_needed();
2433    } else {
2434      n_flushed = page_cleaner_do_flush_batch(
2435              PCT_IO(100),
2436              LSN_MAX);
2437
2438      if (n_flushed) {
2439        MONITOR_INC_VALUE_CUMULATIVE(
2440          MONITOR_FLUSH_BACKGROUND_TOTAL_PAGE,
2441          MONITOR_FLUSH_BACKGROUND_COUNT,
2442          MONITOR_FLUSH_BACKGROUND_PAGES,
2443          n_flushed);
2444      }
2445    }
2446  }
```

Main loop - run until shutdown

check server activity - sleep if server too busy

Something has been changed - do LRU flush and flush from flush_list if needed

nothing has been changed - just do flush from flush_list

background flusher - batch flush LRU

- `buf/buf0flu.cc:2069, 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
  - configurable : innodb_LRU_scan_depth

background flusher - batch flush LRU

- buf/buf0flu.cc:2069, buf_flush_LRU_tail()

```
2068 ulint
2069 buf_flush_LRU_tail(void)
2070 /*====================*/
2071 {
2072   ulint total_flushed = 0;
2073
2074   for (ulint i = 0; i < srv_buf_pool_instances; i++) {
2075
2076     buf_pool_t* buf_pool = buf_pool_from_array(i);
2077     ulint   scan_depth;
2078
2079     /* srv_LRU_scan_depth can be arbitrarily large value.
2080     We cap it with current LRU size. */
2081     buf_pool_mutex_enter(buf_pool);
2082     scan_depth = UT_LIST_GET_LEN(buf_pool->LRU);
2083     buf_pool_mutex_exit(buf_pool);
2084
2085     scan_depth = ut_min(srv_LRU_scan_depth, scan_depth);
```

background flusher - batch flush LRU

- buf/buf0flu.cc:2069, buf_flush_LRU_tail()

```
2087    /* We divide LRU flush into smaller chunks because
2088    there may be user threads waiting for the flush to
2089    end in buf_LRU_get_free_block(). */
2090    for (ulint j = 0;
2091         j < scan_depth;
2092         j += PAGE_CLEANER_LRU_BATCH_CHUNK_SIZE) {
2093
2094      ulint n_flushed = 0;
2095
2096      /* Currently page_cleaner is the only thread
2097      that can trigger an LRU flush. It is possible
2098      that a batch triggered during last iteration is
2099      still running, */
2100      if (buf_flush_LRU(buf_pool,
2101            PAGE_CLEANER_LRU_BATCH_CHUNK_SIZE,
2102            &n_flushed)) {
2103
2104        /* Allowed only one batch per
2105        buffer pool instance. */
2106        buf_flush_wait_batch_end(
2107          buf_pool, BUF_FLUSH_LRU);
2108      }
2109
2110      if (n_flushed) {
2111        total_flushed += n_flushed;
2112      } else {
2113        /* Nothing to flush */
2114        break;
2115      }
2116    }
2117  }
```

PAGE_CLEANER_LRU_BATCH_CHUNK_SIZE : 100

buf_flush_LRU() : batch LRU flush

background flusher - batch flush LRU

- buf/buf0flu.cc:1844, buf_flush_LRU()

```
1843 bool
1844 buf_flush_LRU(
1845 /*==========*/
1846   buf_pool_t* buf_pool, /*!< in/out: buffer pool instance */
1847   ulint   min_n,    /*!< in: wished minimum mumber of blocks
1848           flushed (it is not guaranteed that the
1849           actual number is that big, though) */
1850   ulint*    n_processed)  /*!< out: the number of pages
1851           which were processed is passed
1852           back to caller. Ignored if NULL */
1853 {
1854   ulint   page_count;
1855
1856   if (n_processed) {
1857     *n_processed = 0;
1858   }
1859
1860   if (!buf_flush_start(buf_pool, BUF_FLUSH_LRU)) {
1861     return(false);
1862   }
1863
1864   page_count = buf_flush_batch(buf_pool, BUF_FLUSH_LRU, min_n, 0);
1865
1866   buf_flush_end(buf_pool, BUF_FLUSH_LRU);
1867
1868   buf_flush_common(BUF_FLUSH_LRU, page_count);
1869
1870   if (n_processed) {
1871     *n_processed = page_count;
1872   }
1873
1874   return(true);
1875 }
```

set start lru flushing

do batch flushing

background flusher - batch flush LRU

- buf/buf0flu.cc:1671, buf_flush_bath()

```
1670 ulint
1671 buf_flush_batch(
1672 /*=============*/
1673   buf_pool_t* buf_pool, /*!< in: buffer pool instance */
1674   buf_flush_t flush_type, /*!< in: BUF_FLUSH_LRU or
1675           BUF_FLUSH_LIST; if BUF_FLUSH_LIST,
1676           then the caller must not own any
1677           latches on pages */
1678   ulint   min_n,    /*!< in: wished minimum mumber of blocks
1679           flushed (it is not guaranteed that the
1680           actual number is that big, though) */
1681   lsn_t   lsn_limit)  /*!< in: in the case of BUF_FLUSH_LIST
1682           all blocks whose oldest_modification is
1683           smaller than this should be flushed
1684           (if their number does not exceed
1685           min_n), otherwise ignored */
1686 {
1687   ulint   count = 0;
1688
1689   ut_ad(flush_type == BUF_FLUSH_LRU || flush_type == BUF_FLUSH_LIST);
1690 #ifdef UNIV_SYNC_DEBUG
1691   ut_ad((flush_type != BUF_FLUSH_LIST)
1692         || sync_thread_levels_empty_except_dict());
1693 #endif /* UNIV_SYNC_DEBUG */
1694
1695   buf_pool_mutex_enter(buf_pool);
```

background flusher - batch flush LRU

- buf/buf0flu.cc:1671, buf_flush_bath()

```
1697   /* Note: The buffer pool mutex is released and reacquired within
1698   the flush functions. */
1699   switch (flush_type) {
1700   case BUF_FLUSH_LRU:
1701     count = buf_do_LRU_batch(buf_pool, min_n);
1702     break;
1703   case BUF_FLUSH_LIST:
1704     count = buf_do_flush_list_batch(buf_pool, min_n, lsn_limit);
1705     break;
1706   default:
1707     ut_error;
1708   }
1709
1710   buf_pool_mutex_exit(buf_pool);
1711
1712 #ifdef UNIV_DEBUG
1713   if (buf_debug_prints && count > 0) {
1714     fprintf(stderr, flush_type == BUF_FLUSH_LRU
1715       ? "Flushed %lu pages in LRU flush\n"
1716       : "Flushed %lu pages in flush list flush\n",
1717       (ulong) count);
1718   }
1719 #endif /* UNIV_DEBUG */
1720
1721   return(count);
1722 }
```

do_LRU_batch()

background flusher - batch flush LRU

- buf/buf0flu.cc:1555, buf_do_LRU_batch()

```
1553 static
1554 ulint
1555 buf_do_LRU_batch(
1556 /*===============*/
1557   buf_pool_t* buf_pool, /*!< in: buffer pool instance */
1558   ulint   max)    /*!< in: desired number of
1559           blocks in the free_list */
1560 {
1561   ulint count = 0;
1562
1563   if (buf_LRU_evict_from_unzip_LRU(buf_pool)) {
1564     count += buf_free_from_unzip_LRU_list_batch(buf_pool, max);
1565   }
1566
1567   if (max > count) {
1568     count += buf_flush_LRU_list_batch(buf_pool, max - count);
1569   }
1570
1571   return(count);
1572 }
```

LRU_list_batch()

background flusher - batch flush LRU

- buf/buf0flu.cc:1438, buf_flush_LRU_list_batch()

```
1437 ulint
1438 buf_flush_LRU_list_batch(
1439 /*=======================*/
1440   buf_pool_t* buf_pool, /*!< in: buffer pool instance */
1441   ulint   max)    /*!< in: desired number of
1442           blocks in the free_list */
1443 {
1444   buf_page_t* bpage;
1445   ulint   count = 0;
1446   ulint   scanned = 0;
1447   ulint   free_len = UT_LIST_GET_LEN(buf_pool->free);
1448   ulint   lru_len = UT_LIST_GET_LEN(buf_pool->LRU);
1449
1450   ut_ad(buf_pool_mutex_own(buf_pool));
1451
1452   bpage = UT_LIST_GET_LAST(buf_pool->LRU);
```

get the last buffer from LRU

background flusher - batch flush LRU

- buf/buf0flu.cc:1438, buf_flush_LRU_list_batch()

```
1453   while (bpage != NULL && count < max
1454          && free_len < srv_LRU_scan_depth
1455          && lru_len > BUF_LRU_MIN_LEN) {
1456
1457     ib_mutex_t* block_mutex = buf_page_get_mutex(bpage);
1458     ibool  evict;
1459
1460     mutex_enter(block_mutex);
1461     evict = buf_flush_ready_for_replace(bpage);
1462     mutex_exit(block_mutex);
1463
1464     ++scanned;
1465
1466     /* If the block is ready to be re
1467     free it i.e.: put it on the free
1468     Otherwise we try to flush the blo
1469     neighbors. In this case we'll put
1470     free list in the next pass. We do
1471     of putting blocks to the free lis
1472     just flushing them because after
1473     we have to restart the scan from
1474     the LRU list and if we don't clea
1475     of the flushed pages then the sca
1476     O(n*n). */
```

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 }
```

background flusher - batch flush LRU

- buf/buf0flu.cc:1438, buf_flush_LRU_list_batch()

```
1477      if (evict) {
1478        if (buf_LRU_free_page(bpage, true)) {
1479          /* buf_pool->mutex was potentially
1480          released and reacquired. */
1481          bpage = UT_LIST_GET_LAST(buf_pool->LRU);
1482        } else {
1483          bpage = UT_LIST_GET_PREV(LRU, bpage);
1484        }
```

background flusher - batch flush LRU

- buf/buf0flu.cc:1438, buf_flush_LRU_list_batch()

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

background flusher - batch flush LRU

- buf/buf0flu.cc:1438, buf_flush_LRU_list_batch()

```
1502        if (!buf_flush_page_and_try_neighbors(
1503          bpage, BUF_FLUSH_LRU, max, &count)) {
1504
1505          bpage = prev_bpage;
1506        } else {
1507          /* buf_pool->mutex was released.
1508          reposition the iterator. Note: the
1509          prev block could have been repositioned
1510          too but that should be rare. */
1511
1512          if (prev_bpage != NULL) {
1513
1514            ut_ad(space != ULINT_UNDEFINED);
1515            ut_ad(offset != ULINT_UNDEFINED);
1516
1517            prev_bpage = buf_page_hash_get(
1518              buf_pool, space, offset);
1519          }
1520
1521          bpage = prev_bpage;
1522        }
1523      }
1524
1525      free_len = UT_LIST_GET_LEN(buf_pool->free);
1526      lru_len = UT_LIST_GET_LEN(buf_pool->LRU);
1527    }
```

background flusher - batch flush LRU

- buf/buf0flu.cc:1313, buf_flush_page_and_try_neighbors()

```
1312 ibool
1313 buf_flush_page_and_try_neighbors(
1314 /*================================*/
1315   buf_page_t* bpage,    /*!< in: buffer control block,
1316           must be
1317           buf_page_in_file(bpage) */
1318   buf_flush_t flush_type, /*!< in: BUF_FLUSH_LRU
1319           or BUF_FLUSH_LIST */
1320   ulint   n_to_flush, /*!< in: number of pages to
1321           flush */
1322   ulint*    count)    /*!< in/out: number of pages
1323           flushed */
1324 {
1325   ibool   flushed;
1326   ib_mutex_t* block_mutex;
1327 #ifdef UNIV_DEBUG
1328   buf_pool_t* buf_pool = buf_pool_from_bpage(bpage);
1329 #endif /* UNIV_DEBUG */
1330
1331   ut_ad(buf_pool_mutex_own(buf_pool));
1332
1333   block_mutex = buf_page_get_mutex(bpage);
1334   mutex_enter(block_mutex);
```

background flusher - batch flush LRU

- `buf/buf0flu.cc:1313, buf_flush_page_and_try_neighbors()`

```
1338  if (buf_flush_ready_for_flush(bpage, flush_type)) {
1339    buf_pool_t* buf_pool;
1340
1341    buf_pool = buf_pool_from_bpage(bpage);
1342
1343    buf_pool_mutex_exit(buf_pool);
1344
1345    /* These fields are protected by both the
1346    buffer pool mutex and block mutex. */
1347    ulint space = buf_page_get_space(bpage);
1348    ulint offset = buf_page_get_page_no(bpage);
1349
1350    mutex_exit(block_mutex);
1351
1352    /* Try to flush also all the neighbors */
1353    *count += buf_flush_try_neighbors(
1354      space, offset, flush_type, *count, n_to_flush);
1355
1356    buf_pool_mutex_enter(buf_pool);
1357
1358    flushed = TRUE;
```

flush a page with neighbors

background flusher - batch flush LRU

- buf/buf0flu.cc:1163, buf_flush_try_neighbors()

```
1162 ulint
1163 buf_flush_try_neighbors(
1164 /*=====================*/
1165   ulint   space,    /*!< in: space id */
1166   ulint   offset,   /*!< in: page offset */
1167   buf_flush_t flush_type, /*!< in: BUF_FLUSH_LRU or
1168           BUF_FLUSH_LIST */
1169   ulint   n_flushed,  /*!< in: number of pages
1170           flushed so far in this batch */
1171   ulint   n_to_flush) /*!< in: maximum number of pages
1172           we are allowed to flush */
1173 {
1174   ulint   i;
1175   ulint   low;
1176   ulint   high;
1177   buf_pool_t* buf_pool = buf_pool_get(space, offset);
1178
1179   ut_ad(flush_type == BUF_FLUSH_LRU || flush_type == BUF_FLUSH_LIST);
```

background flusher - batch flush LRU

- buf/buf0flu.cc:1163, buf_flush_try_neighbors()

```
1181   if (UT_LIST_GET_LEN(buf_pool->LRU) < BUF_LRU_OLD_MIN_LEN
1182       || srv_flush_neighbors == 0) {
1183     /* If there is little space or neighbor flushing is
1184     not enabled then just flush the victim. */
1185     low = offset;
1186     high = offset + 1;
```

no flush neighbor

skip check flush neighbor

background flusher - batch flush LRU

- buf/buf0flu.cc:1163, buf_flush_try_neighbors()

```
1269    if (flush_type != BUF_FLUSH_LRU
1270        || i == offset
1271        || buf_page_is_old(bpage)) {
1272
1273      ib_mutex_t* block_mutex = buf_page_get_mutex(bpage);
1274
1275      mutex_enter(block_mutex);
1276
1277      if (buf_flush_ready_for_flush(bpage, flush_type)
1278          && (i == offset || bpage->buf_fix_count == 0)
1279          && buf_flush_page(
1280          buf_pool, bpage, flush_type, false)) {
1281
1282        ++count;
1283
1284        continue;
1285      }
```

flush page, but no sync

background flusher - flush page

- `buf/buf0flu.cc:993, buf_flush_page()`
- Writes a flushable page asynchronously from the buffer pool to a file
  - set io_fix : BUF_IO_WRITE
  - set flush event type
  - **add to batch write**
    - write block in single flush case

background flusher - flush page

- buf/buf0flu.cc:993, buf_flush_page()

```
 992 bool
 993 buf_flush_page(
 994 /*===========*/
 995   buf_pool_t* buf_pool, /*!< in: buffer pool instance */
 996   buf_page_t* bpage,    /*!< in: buffer control block */
 997   buf_flush_t flush_type, /*!< in: type of flush */
 998   bool    sync)   /*!< in: true if sync IO request */
 999 {
1000   ut_ad(flush_type < BUF_FLUSH_N_TYPES);
1001   ut_ad(buf_pool_mutex_own(buf_pool));
1002   ut_ad(buf_page_in_file(bpage));
1003   ut_ad(!sync || flush_type == BUF_FLUSH_SINGLE_PAGE);
1004
1005   ib_mutex_t* block_mutex = buf_page_get_mutex(bpage);
1006
1007   ut_ad(mutex_own(block_mutex));
1008
1009   ut_ad(buf_flush_ready_for_flush(bpage, flush_type));
1010
1011   bool            is_uncompressed;
1012
1013   is_uncompressed = (buf_page_get_state(bpage) == BUF_BLOCK_FILE_PAGE);
1014   ut_ad(is_uncompressed == (block_mutex != &buf_pool->zip_mutex));
1015
1016   ibool           flush;
1017   rw_lock_t*  rw_lock;
1018   bool            no_fix_count = bpage->buf_fix_count == 0;
1019
```

status check

background flusher - flush page

- buf/buf0flu.cc:993, buf_flush_page()

```
1020  if (!is_uncompressed) {
1021    flush = TRUE;
1022    rw_lock = NULL;
1023
1024  } else if (!(no_fix_count || flush_type == BUF_FLUSH_LIST)) {
1025    /* This is a heuristic, to avoid expensive S attempts. */
1026    flush = FALSE;
1027  } else {
1028
1029    rw_lock = &reinterpret_cast<buf_block_t*>(bpage)->lock;
1030
1031    if (flush_type != BUF_FLUSH_LIST) {
1032      flush = rw_lock_s_lock_gen_nowait(
1033        rw_lock, BUF_IO_WRITE);
1034    } else {
1035      /* Will S lock later */
1036      flush = TRUE;
1037    }
1038  }
```

get FIX LOCK

background flusher - flush page

- buf/buf0flu.cc:993, buf_flush_page()

```
1040   if (flush) {
1041
1042     /* We are committed to flushing by the time we get here */
1043
1044     buf_page_set_io_fix(bpage, BUF_IO_WRITE);
1045
1046     buf_page_set_flush_type(bpage, flush_type);
1047
1048     if (buf_pool->n_flush[flush_type] == 0) {
1049
1050       os_event_reset(buf_pool->no_flush[flush_type]);
1051     }
1052
1053     ++buf_pool->n_flush[flush_type];
1054
1055     mutex_exit(block_mutex);
1056     buf_pool_mutex_exit(buf_pool);
```

set fix and flush type

background flusher - flush page

- buf/buf0flu.cc:993, buf_flush_page()

```
1058     if (flush_type == BUF_FLUSH_LIST
1059         && is_uncompressed
1060         && !rw_lock_s_lock_gen_nowait(rw_lock, BUF_IO_WRITE)) {
1061       /* avoiding deadlock possibility involves doublewrite
1062          buffer, should flush it, because it might hold the
1063          another block->lock. */
1064       buf_dblwr_flush_buffered_writes();
1065
1066       rw_lock_s_lock_gen(rw_lock, BUF_IO_WRITE);
1067     }
1068
1069     /* Even though bpage is not protected by any mutex at this
1070        point, it is safe to access bpage, because it is io_fixed and
1071        oldest_modification != 0.  Thus, it cannot be relocated in the
1072        buffer pool or removed from flush_list or LRU_list. */
1073
1074     buf_flush_write_block_low(bpage, flush_type, sync);
1075   }
1076
1077   return(flush);
1078 }
```

background flusher - flush page

- buf/buf0flu.cc:877, buf_flush_write_block_low()

```
876 void
877 buf_flush_write_block_low(
878 /*=======================*/
879   buf_page_t* bpage,    /*!< in: buffer block to write */
880   buf_flush_t flush_type, /*!< in: type of flush */
881   bool    sync)   /*!< in: true if sync IO request */
882 {
883   ulint zip_size  = buf_page_get_zip_size(bpage);
884   page_t* frame   = NULL;
922   /* Force the log to the disk before writing the modified block */
923   log_write_up_to(bpage->newest_modification, LOG_WAIT_ALL_GROUPS, TRUE);
924 #endif
```

flush log (Transaction Log - WAL)

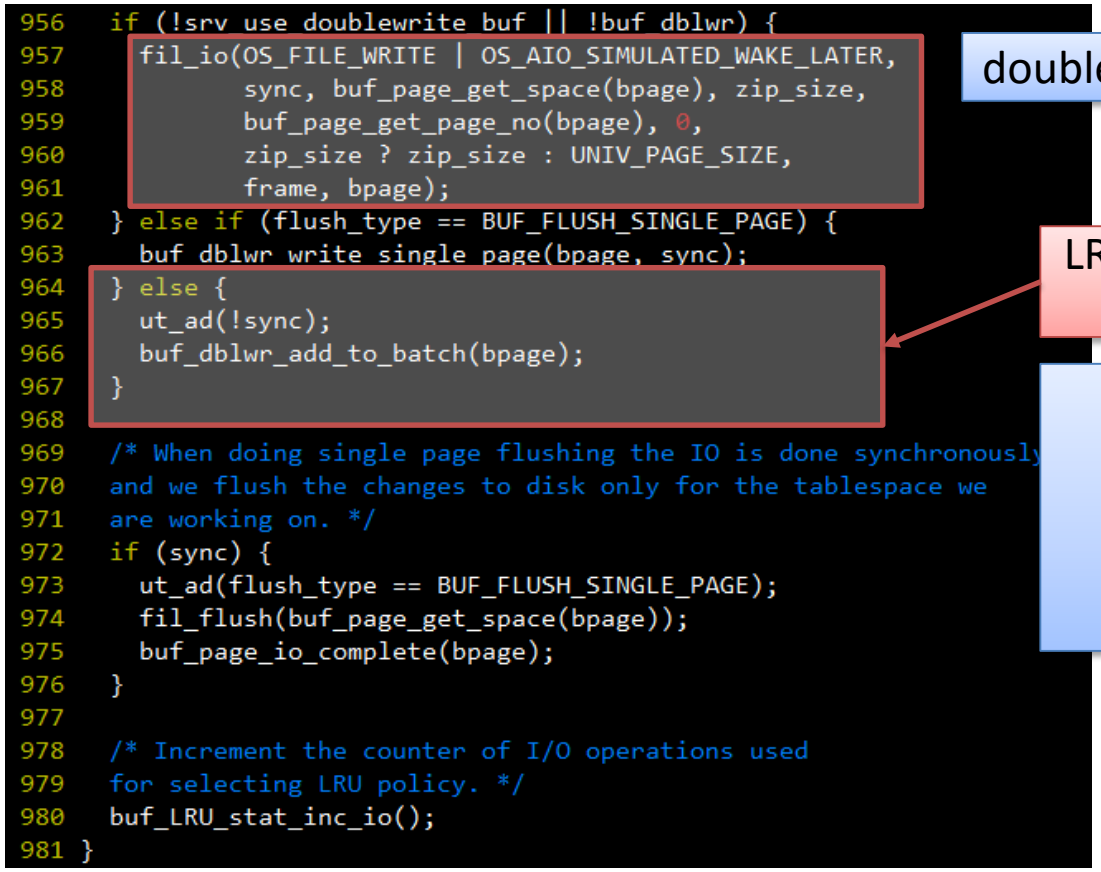
background flusher - flush page

- buf/buf0flu.cc:877, buf_flush_write_block_low()

```
925   switch (buf_page_get_state(bpage)) {
926   case BUF_BLOCK_POOL_WATCH:
927   case BUF_BLOCK_ZIP_PAGE: /* The page should be dirty. */
928   case BUF_BLOCK_NOT_USED:
929   case BUF_BLOCK_READY_FOR_USE:
930   case BUF_BLOCK_MEMORY:
931   case BUF_BLOCK_REMOVE_HASH:
932     ut_error;
933     break;
934   case BUF_BLOCK_ZIP_DIRTY:
935     frame = bpage->zip.data;
936
937     ut_a(page_zip_verify_checksum(frame, zip_size));
938
939     mach_write_to_8(frame + FIL_PAGE_LSN,
940         bpage->newest_modification);
941     memset(frame + FIL_PAGE_FILE_FLUSH_LSN, 0, 8);
942     break;
943   case BUF_BLOCK_FILE_PAGE:
944     frame = bpage->zip.data;
945     if (!frame) {
946       frame = ((buf_block_t*) bpage)->frame;
947     }
948
949     buf_flush_init_for_writing(((buf_block_t*) bpage)->frame,
950             bpage->zip.data
951             ? &bpage->zip : NULL,
952             bpage->newest_modification);
953     break;
954   }
```

background flusher - flush page

- buf/buf0flu.cc:877, buf_flush_write_block_low()



```
956   if (!srv_use_doublewrite_buf || !buf_dblwr) {
957     fil_io(OS_FILE_WRITE | OS_AIO_SIMULATED_WAKE_LATER,
958            sync, buf_page_get_space(bpage), zip_size,
959            buf_page_get_page_no(bpage), 0,
960            zip_size ? zip_size : UNIV_PAGE_SIZE,
961            frame, bpage);
962   } else if (flush_type == BUF_FLUSH_SINGLE_PAGE) {
963     buf_dblwr_write_single_page(bpage, sync);
964   } else {
965     ut_ad(!sync);
966     buf_dblwr_add_to_batch(bpage);
967   }
968
969   /* When doing single page flushing the IO is done synchronously
970   and we flush the changes to disk only for the tablespace we
971   are working on. */
972   if (sync) {
973     ut_ad(flush_type == BUF_FLUSH_SINGLE_PAGE);
974     fil_flush(buf_page_get_space(bpage));
975     buf_page_io_complete(bpage);
976   }
977
978   /* Increment the counter of I/O operations used
979   for selecting LRU policy. */
980   buf_LRU_stat_inc_io();
981 }
```

background flusher - batch flush LRU

- buf/buf0flu.cc:1844, buf_flush_LRU()

```
1843 bool
1844 buf_flush_LRU(
1845 /*==========*/
1846   buf_pool_t* buf_pool, /*!< in/out: buffer pool instance */
1847   ulint   min_n,    /*!< in: wished minimum number of blocks
1848           flushed (it is not guaranteed that the
1849           actual number is that big, though) */
1850   ulint*    n_processed)  /*!< out: the number of pages
1851           which were processed is passed
1852           back to caller. Ignored if NULL */
1853 {
1854   ulint   page_count;
1855
1856   if (n_processed) {
1857     *n_processed = 0;
1858   }
1859
1860   if (!buf_flush_start(buf_pool, BUF_FLUSH_LRU)) {
1861     return(false);
1862   }
1863
1864   page_count = buf_flush_batch(buf_pool, BUF_FLUSH_LRU, min_n, 0);
1865
1866   buf_flush_end(buf_pool, BUF_FLUSH_LRU);
1867
1868   buf_flush_common(BUF_FLUSH_LRU, page_count);
1869
1870   if (n_processed) {
1871     *n_processed = page_count;
1872   }
1873
1874   return(true);
1875 }
```

we just finished this

do flush common

background flusher - batch flush LRU

- buf/buf0flu.cc:1728, buf_flush_common()

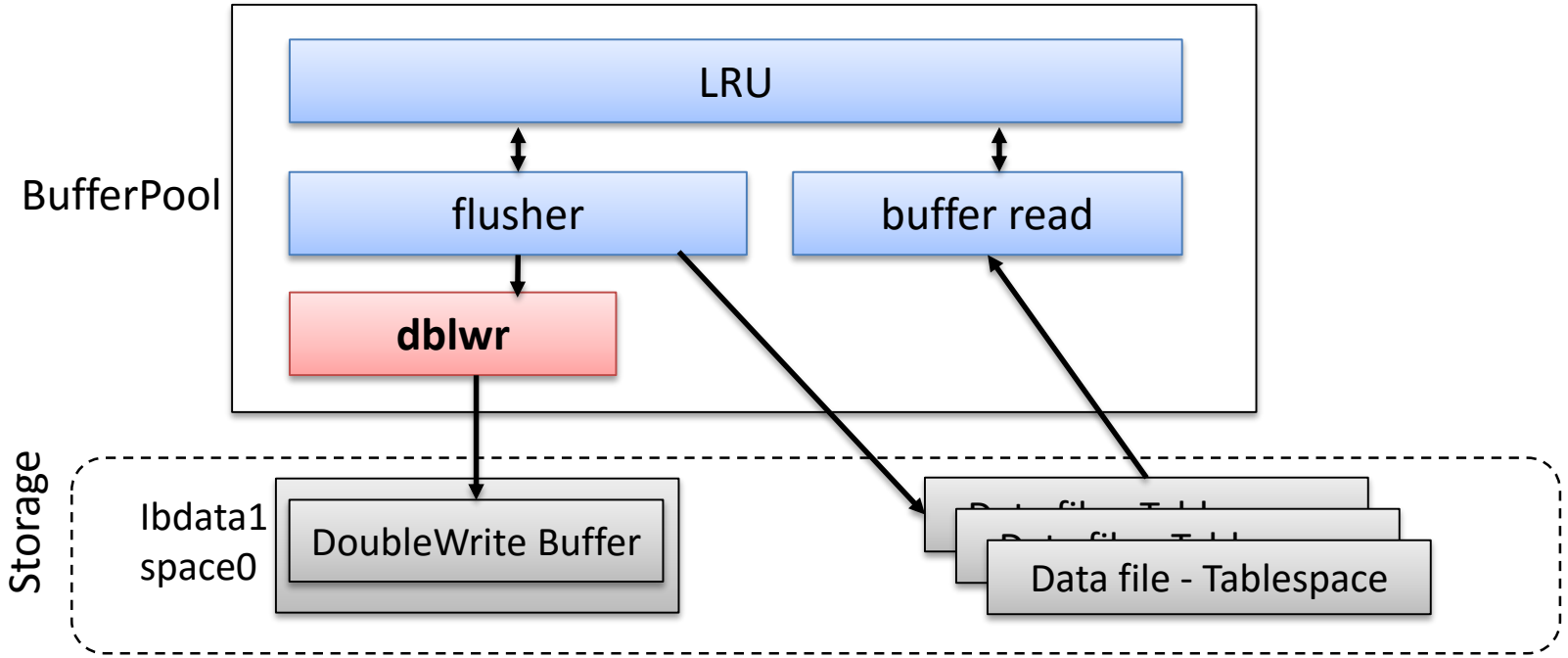
```
1724 /**********************************************************************//**
1725 Gather the aggregated stats for both flush list and LRU list flushing */
1726 static
1727 void
1728 buf_flush_common(
1729 /*=============*/
1730   buf_flush_t flush_type, /*!< in: type of flush */
1731   ulint   page_count) /*!< in: number of pages flushed */
1732 {
1733   buf_dblwr_flush_buffered_writes();
1734
1735   ut_a(flush_type == BUF_FLUSH_LRU || flush_type == BUF_FLUSH_LIST);
1736
1737 #ifdef UNIV_DEBUG
1738   if (buf_debug_prints && page_count > 0) {
1739     fprintf(stderr, flush_type == BUF_FLUSH_LRU
1740       ? "Flushed %lu pages in LRU flush\n"
1741       : "Flushed %lu pages in flush list flush\n",
1742       (ulong) page_count);
1743   }
1744 #endif /* UNIV_DEBUG */
1745
1746   srv_stats.buf_pool_flushed.add(page_count);
1747 }
```

flush all buffers we gathered so far, write the buffers to dblwr area first, then issue it to datafile : **see this later**

DBLWR

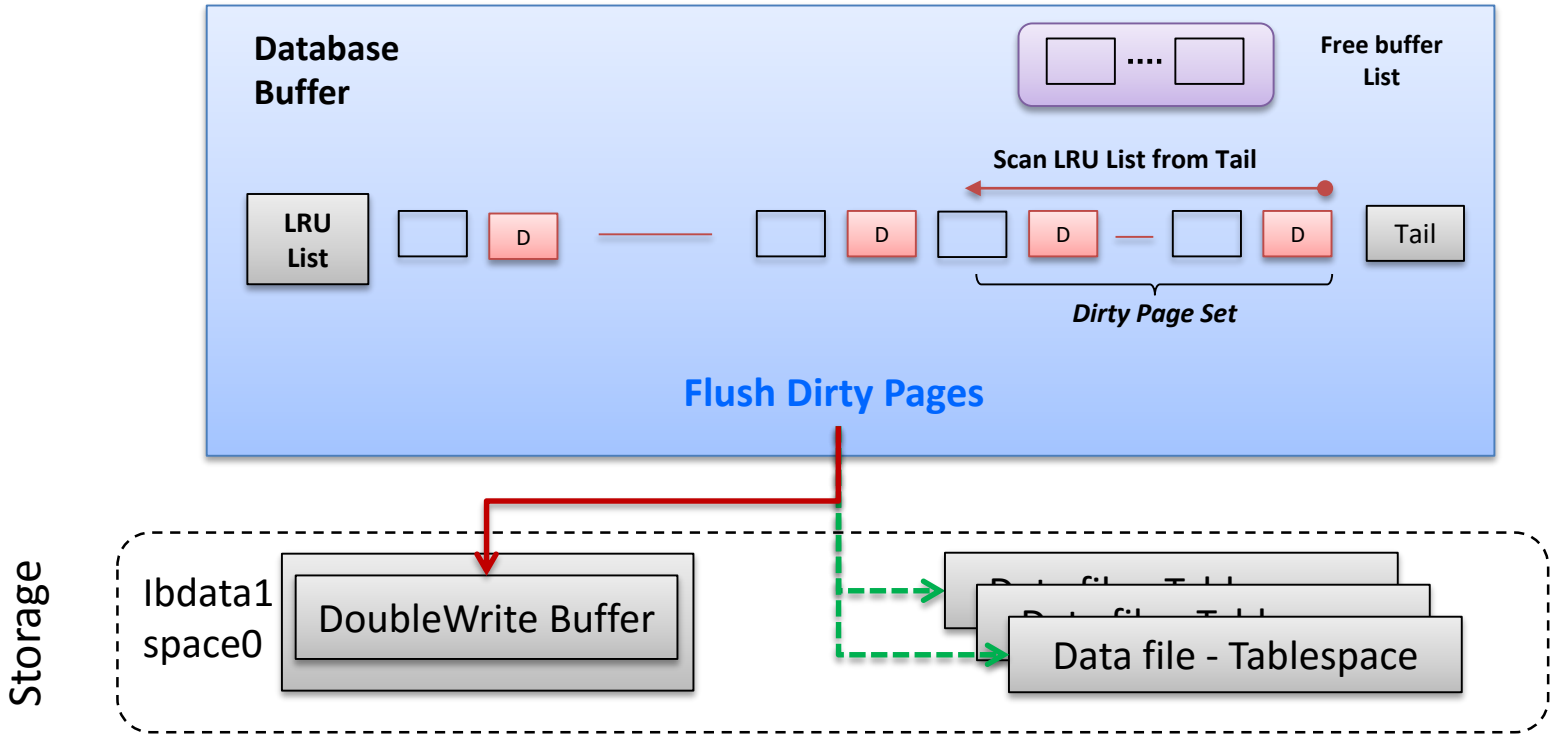
Overview

- Buffer Manager
  - Buffer Pool (buf0buf.cc) : Buffer Pool manager
  - Buffer Read (buf0read.cc) : Read Buffer
  - LRU (buf0lru.cc) : Buffer Replacement
  - Flusher (buf0flu.cc) : dirty page writer, background flusher
  - **Doublewrite (buf0dblwr.cc) : Doublewrite**



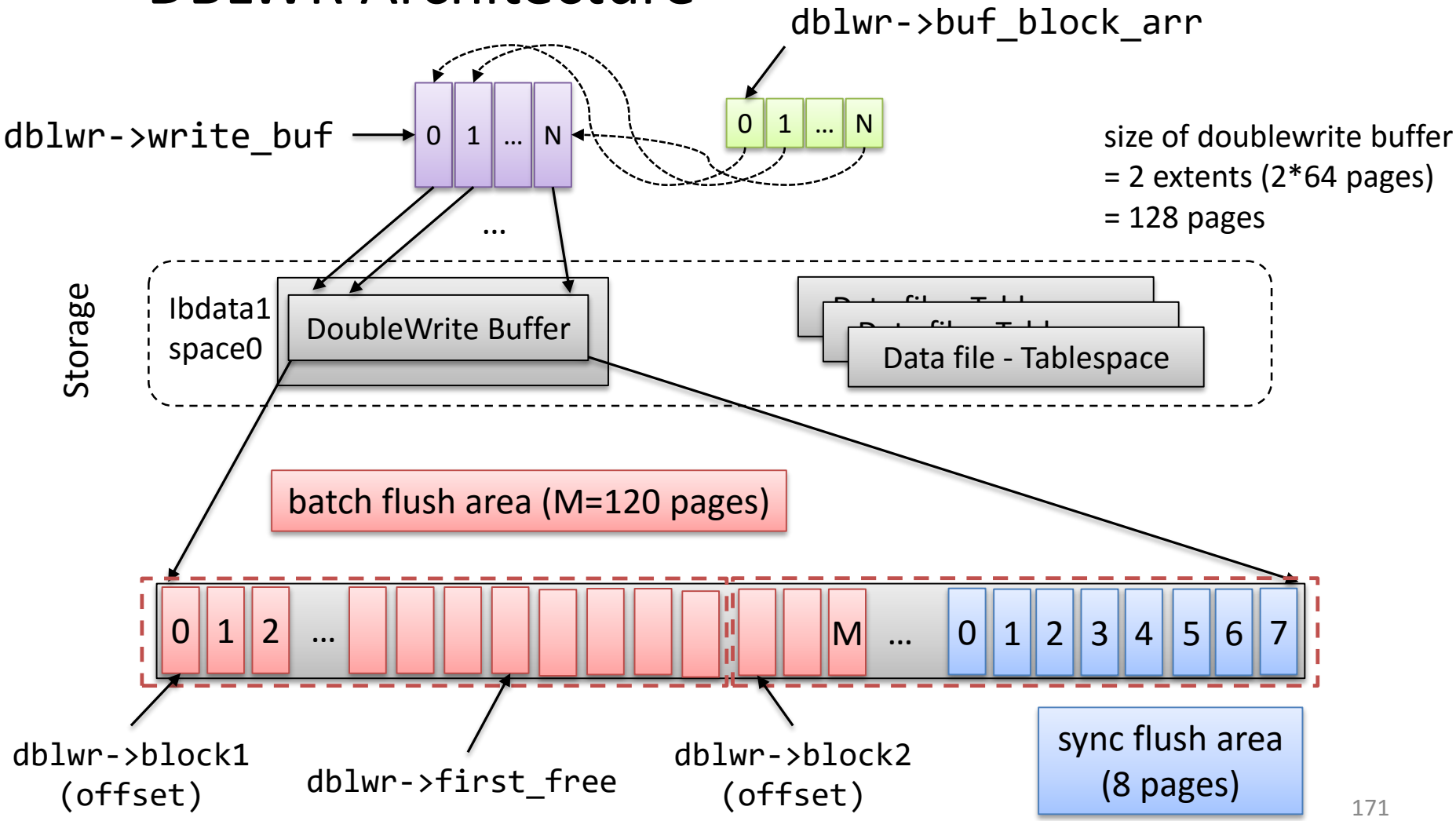
Double Write Buffer

- To avoid torn page written problem
- Write dirty pages special storage area in system table space priori to write database file



Double Write Buffer

- DBLWR Architecture



Double Write Buffer

- Double write control struct

```
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. */
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 */
157 };
```

dblwr flush - single page

- buf/buf0flu.cc:877, buf_flush_write_block_low()

```
956   if (!srv_use_doublewrite_buf || !buf_dblwr) {
957     fil_io(OS_FILE_WRITE | OS_AIO_SIMULATED_WAKE_LATER,
958            sync, buf_page_get_space(bpage), zip_size,
959            buf_page_get_page_no(bpage), 0,
960            zip_size ? zip_size : UNIV_PAGE_SIZE,
961            frame, bpage);
962   } else if (flush_type == BUF_FLUSH_SINGLE_PAGE) {
963     buf_dblwr_write_single_page(bpage, sync);
964   } else {
965     ut_ad(!sync);
966     buf_dblwr_add_to_batch(bpage);
967   }
968
969   /* When doing single page flushing the IO is done synchronously
970   and we flush the changes to disk only for the tablespace we
971   are working on. */
972   if (sync) {
973     ut_ad(flush_type == BUF_FLUSH_SINGLE_PAGE);
974     fil_flush(buf_page_get_space(bpage));
975     buf_page_io_complete(bpage);
976   }
977
978   /* Increment the counter of I/O operations used
979   for selecting LRU policy. */
980   buf_LRU_stat_inc_io();
981 }
```

do single page double write then write to datafile

sync buffered write to disk : call fsync by **fil_flush**

dblwr flush - single page

- buf/buf0dblwr.cc:1055, buf_dblwr_write_single_page()

```
1054 void
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 {
1060   ulint   n_slots;
1061   ulint   size;
1062   ulint   zip_size;
1063   ulint   offset;
1064   ulint   i;
1065
1066   ut_a(buf_page_in_file(bpage));
1067   ut_a(srv_use_doublewrite_buf);
1068   ut_a(buf_dblwr != NULL);
1069
1070   /* total number of slots available for single page flushes
1071   starts from srv_doublewrite_batch_size to the end of the
1072   buffer. */
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 in single flush case =
2 * TRX_SYS_DOUBLEWRITE_BLOCK_SIZE
- BATCH_SIZE
= 128-120
= 8

dblwr flush - single page

- buf/buf0dblwr.cc:1055, buf_dblwr_write_single_page()

```
1077   if (buf_page_get_state(bpage) == BUF_BLOCK_FILE_PAGE) {
1078
1079     /* Check that the actual page in the buffer pool is
1080     not corrupt and the LSN values are sane. */
1081     buf_dblwr_check_block((buf_block_t*) bpage);
1082
1083     /* Check that the page as written to the doublewrite
1084     buffer has sane LSN values. */
1085     if (!bpage->zip.data) {
1086       buf_dblwr_check_page_lsn(
1087         ((buf_block_t*) bpage)->frame);
1088     }
1089   }
```

check page status especially LSN

dblwr flush - single page

- buf/buf0dblwr.cc:1055, 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   }
```

There is not enough space left, wait until current dblwr done.

find free slot in single flush area in dblwr

dblwr flush - single page

- buf/buf0dblwr.cc:1055, buf_dblwr_write_single_page()

```
1111  /* We are guaranteed to find a slot. */
1112  ut_a(i < size);
1113  buf_dblwr->in_use[i] = true;
1114  buf_dblwr->s_reserved++;
1115  buf_dblwr->buf_block_arr[i] = bpage;
1116
1117  /* increment the doublewrite flushed pages counter */
1118  srv_stats.dblwr_pages_written.inc();
1119  srv_stats.dblwr_writes.inc();
1120
1121  mutex_exit(&buf_dblwr->mutex);
1122
1123  /* Lets see if we are going to write in the first or second
1124  block of the doublewrite buffer. */
1125  if (i < TRX_SYS_DOUBLEWRITE_BLOCK_SIZE) {
1126    offset = buf_dblwr->block1 + i;
1127  } else {
1128    offset = buf_dblwr->block2 + i
1129       - TRX_SYS_DOUBLEWRITE_BLOCK_SIZE;
1130  }
```

decide block1 or block2

calculation looks wrong : because we started to check at 120, never take block1

dblwr flush - single page

- buf/buf0dblwr.cc:1055, buf_dblwr_write_single_page()

```
1128   } else {
1129     /* It is a regular page. Write it directly to the
1130     doublewrite buffer */
1131     fil_io(OS_FILE_WRITE, true,
1132            page_id_t(TRX_SYS_SPACE, offset), univ_page_size, 0,
1133            univ_page_size.physical(),
1134            (void*) ((buf_block_t*) bpage)->frame,
1135            NULL);
1136   }
1137
1138   /* Now flush the doublewrite buffer data to disk */
1139   fil_flush(TRX_SYS_SPACE);
1140
1141   /* We know that the write has been flushed to disk now
1142   and during recovery we will find it in the doublewrite buffer
1143   blocks. Next do the write to the intended position. */
1144   buf_dblwr_write_block_to_datafile(bpage, sync);
1145 }
```

write block to dblwr buffer in system table space

sync system tablespace

write to data file

dblwr flush - single page

- buf/buf0dblwr.cc:782, buf_dblwr_write_block_to_datafile()

```
781 void
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 {
788   ut_a(bpage);
789   ut_a(buf_page_in_file(bpage));
790
791   const ulint flags = sync
792     ? OS_FILE_WRITE
793     : OS_FILE_WRITE | OS_AIO_SIMULATED_WAKE_LATER;
794
795   if (bpage->zip.data) {
796     fil_io(flags, sync, buf_page_get_space(bpage),
797            buf_page_get_zip_size(bpage),
798            buf_page_get_page_no(bpage), 0,
799            buf_page_get_zip_size(bpage),
800            (void*) bpage->zip.data,
801            (void*) bpage);
802
803     return;
804   }
805
806
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 : **see this function later**

dblwr flush - batch LRU

- buf/buf0flu.cc:877, buf_flush_write_block_low()

```
956   if (!srv_use_doublewrite_buf || !buf_dblwr) {
957     fil_io(OS_FILE_WRITE | OS_AIO_SIMULATED_WAKE_LATER,
958            sync, buf_page_get_space(bpage), zip_size,
959            buf_page_get_page_no(bpage), 0,
960            zip_size ? zip_size : UNIV_PAGE_SIZE,
961            frame, bpage);
962   } else if (flush_type == BUF_FLUSH_SINGLE_PAGE) {
963     buf_dblwr_write_single_page(bpage, sync);
964   } else {
965     ut_ad(!sync);
966     buf_dblwr_add_to_batch(bpage);
967   }
968
969   /* When doing single page flushing the IO is done synchronously
970   and we flush the changes to disk only for the tablespace we
971   are working on. */
972   if (sync) {
973     ut_ad(flush_type == BUF_FLUSH_SINGLE_PAGE);
974     fil_flush(buf_page_get_space(bpage));
975     buf_page_io_complete(bpage);
976   }
977
978   /* Increment the counter of I/O operations used
979   for selecting LRU policy. */
980   buf_LRU_stat_inc_io();
981 }
```

LRU_batch flush case

dblwr flush - add to batch

- buf/buf0dblwr.cc:968, buf_dblwr_add_to_batch()

```
967 void
968 buf_dblwr_add_to_batch(
969 /*====================*/
970   buf_page_t* bpage)  /*!< in: buffer block to write */
971 {
972   ulint zip_size;
973
974   ut_a(buf_page_in_file(bpage));
975
976 try_again:
977   mutex_enter(&buf_dblwr->mutex);
978
979   ut_a(buf_dblwr->first_free <= srv_doublewrite_batch_size);
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   }
```

another batch is already running - wait until done

dblwr flush - add to batch

- buf/buf0dblwr.cc:968, 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  }
1003
1004  zip_size = buf_page_get_zip_size(bpage);
1005
1006  if (zip_size) {
1007    UNIV_MEM_ASSERT_RW(bpage->zip.data, zip_size);
1008    /* Copy the compressed page and clear the rest. */
1009    memcpy(buf_dblwr->write_buf
1010           + UNIV_PAGE_SIZE * buf_dblwr->first_free,
1011           bpage->zip.data, zip_size);
1012    memset(buf_dblwr->write_buf
1013           + UNIV_PAGE_SIZE * buf_dblwr->first_free
1014           + zip_size, 0, UNIV_PAGE_SIZE - zip_size);
1015  } else {
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  }
```

run out of batch space in doublewrite area - post flush doublewrite buffers

copy current block to dblwr-first_free

dblwr flush - add to batch

- buf/buf0dblwr.cc:968, buf_dblwr_add_to_batch()

```
1025   buf_dblwr->buf_block_arr[buf_dblwr->first_free] = bpage;
1026
1027   buf_dblwr->first_free++;
1028   buf_dblwr->b_reserved++;
1029
1030   ut_ad(!buf_dblwr->batch_running);
1031   ut_ad(buf_dblwr->first_free == buf_dblwr->b_reserved);
1032   ut_ad(buf_dblwr->b_reserved <= srv_doublewrite_batch_size);
1033
1034   if (buf_dblwr->first_free == srv_doublewrite_batch_size) {
1035     mutex_exit(&(buf_dblwr->mutex));
1036
1037     buf_dblwr_flush_buffered_writes();
1038
1039     return;
1040   }
1041
1042   mutex_exit(&(buf_dblwr->mutex));
1043 }
```

dblwr flush - flush buffered write

- buf/buf0dblwr.cc:825, buf_dblwr_flush_buffered_writes()

```
824 void
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
838 try_again:
839   mutex_enter(&buf_dblwr->mutex);
840
841   /* Write first to doublewrite buffer blocks. We use synchronous
842   aio and thus know that file write has been completed when the
843   control returns. */
844
845   if (buf_dblwr->first_free == 0) {
846
847     mutex_exit(&buf_dblwr->mutex);
848
849     return;
850   }
```

doublewrite off case

doublewrite empty

dblwr flush - flush buffered write

- buf/buf0dblwr.cc:825, buf_dblwr_flush_buffered_writes()

```
852  if (buf_dblwr->batch_running) {
853    /* Another thread is running the batch right now. Wait
854    for it to finish. */
855    ib_int64_t  sig_count = os_event_reset(buf_dblwr->b_event);
856    mutex_exit(&buf_dblwr->mutex);
857
858    os_event_wait_low(buf_dblwr->b_event, sig_count);
859    goto try_again;
860  }
861
862  ut_a(!buf_dblwr->batch_running);
863  ut_ad(buf_dblwr->first_free == buf_dblwr->b_reserved);
864
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);
```

another batch running

change batch running status and unlock mutex : no body won't be here except me

dblwr flush - flush buffered write

- buf/buf0dblwr.cc:825, buf_dblwr_flush_buffered_writes()

```
876   write_buf = buf_dblwr->write_buf;
877
878   for (ulint len2 = 0, i = 0;
879        i < buf_dblwr->first_free;
880        len2 += UNIV_PAGE_SIZE, i++) {
881
882     const buf_block_t*  block;
883
884     block = (buf_block_t*) buf_dblwr->buf_block_arr[i];
885
886     if (buf_block_get_state(block) != BUF_BLOCK_FILE_PAGE
887         || block->page.zip.data) {
888       /* No simple validate for compressed
889       pages exists. */
890       continue;
891     }
892
893     /* Check that the actual page in the buffer pool is
894     not corrupt and the LSN values are sane. */
895     buf_dblwr_check_block(block);
896
897     /* Check that the page as written to the doublewrite
898     buffer has sane LSN values. */
899     buf_dblwr_check_page_lsn(write_buf + len2);
900   }
```

check block consistency

dblwr flush - flush buffered write

- buf/buf0dblwr.cc:825, 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   }
914
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);
925
926 flush:
```

issue write op for block1 - synchronous

if current write use only block1 then flush

issue write op for block2 - synchronous

dblwr flush - flush buffered write

- buf/buf0dblwr.cc:825, 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);
933
934   /* We know that the writes have been flushed to disk now
935   and in recovery we will find them in the doublewrite buffer
936   blocks. Next do the writes to the intended positions. */
949   ut_ad(first_free == buf_dblwr->first_free);
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 thread to actually post the
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 tablespace

write blocks to datafile

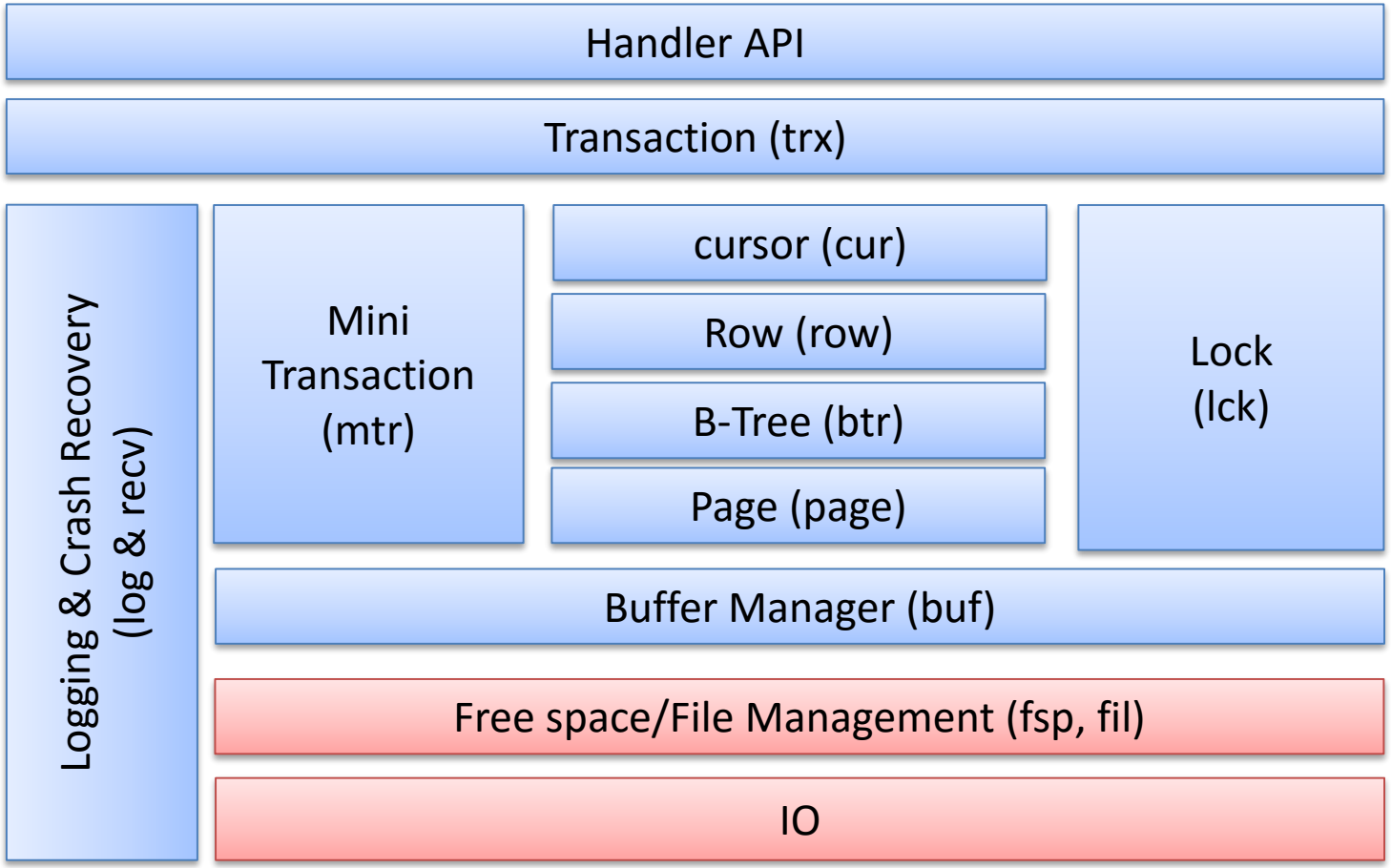
wake io thread

Contents

- Buffer manager
  - Overview
  - Buffer Pool (buf0buf.cc)
  - Buffer Read (buf0read.cc)
  - LRU (buf0lru.cc)
  - Flusher (buf0flu.cc)
  - Doublewrite (buf0dblwr.cc)
- File manager
  - Overview
  - file system (fil0fil.cc)
  - OS specific impl. for file system (os0file.cc)

Overview

- InnoDB Architecture - File management(fil_system)

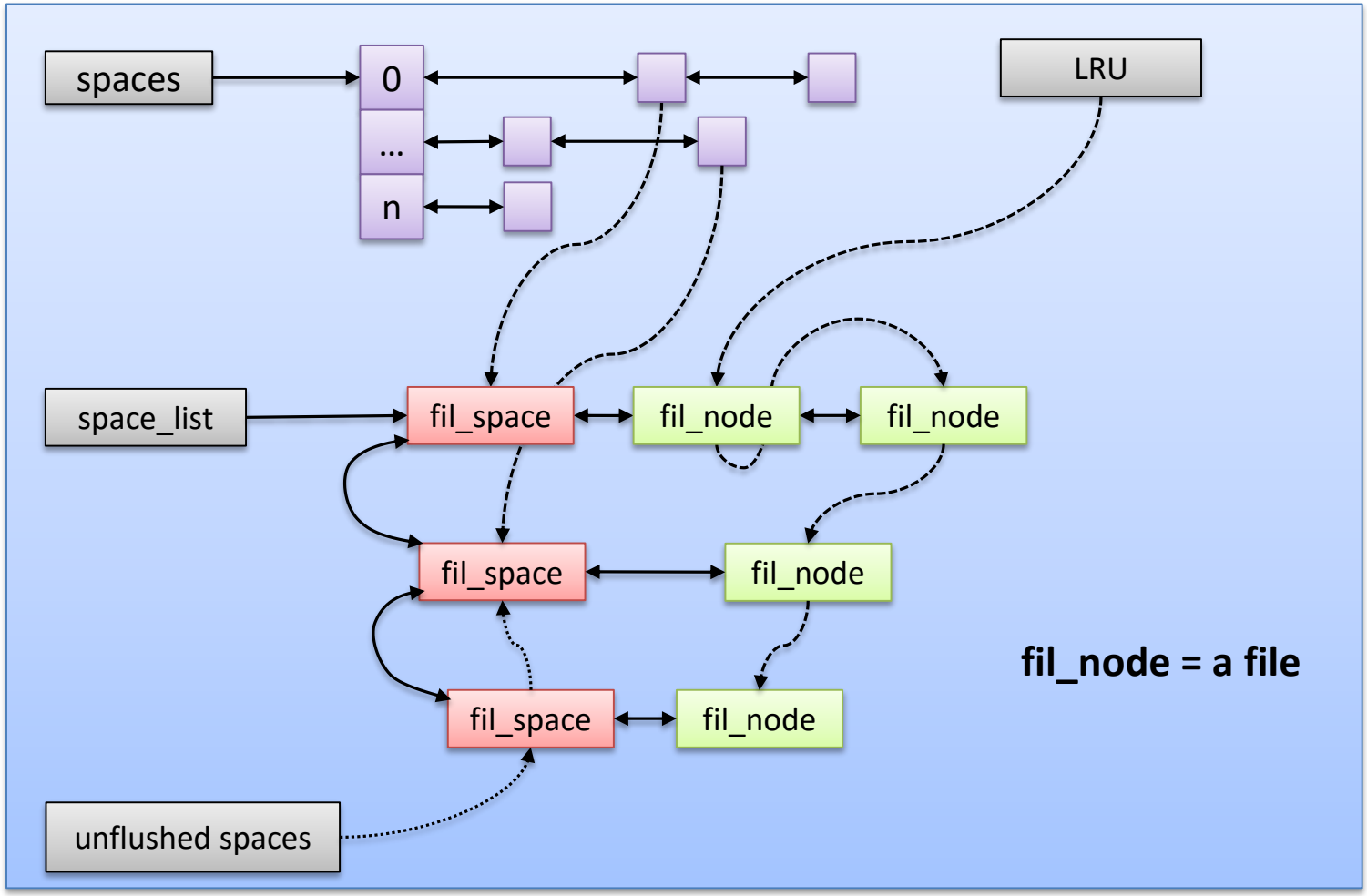


File manager - File system

- File system
  - resides global area in innodb
  - manage open files
  - tablespace and node
- File open/io/flush
  - open a file
  - do io - read/write
  - flush (fsync)

File system overview

- File system



File system structure

- fil_system_t

```
261 struct fil_system_t {
262 #ifndef UNIV_HOTBACKUP
263   ib_mutex_t    mutex;    /*!< The mutex protecting the cache */
264 #endif /* !UNIV_HOTBACKUP */
265   hash_table_t* spaces;   /*!< The hash table of spaces in the
266           system; they are hashed on the space
267           id */
268   hash_table_t* name_hash;  /*!< hash table based on the space
269           name */
270   UT_LIST_BASE_NODE_T(fil_node_t) LRU;
271           /*!< base node for the LRU list of the
272           most recently used open files with no
273           pending i/o's; if we start an i/o on
274           the file, we first remove it from this
275           list, and return it to the start of
276           the list when the i/o ends;
277           log files and the system tablespace are
278           not put to this list: they are opened
279           after the startup, and kept open until
280           shutdown */
281   UT_LIST_BASE_NODE_T(fil_space_t) unflushed_spaces;
282           /*!< base node for the list of those
283           tablespaces whose files contain
284           unflushed writes; those spaces have
285           at least one file node where
286           modification_counter > flush_counter */
287   ulint   n_open;   /*!< number of files currently open */
288   ulint   max_n_open; /*!< n_open is not allowed to exceed
289           this */
290   ib_int64_t  modification_counter;/*!< when we write to a file we
291           increment this by one */
```

```
292   ulint   max_assigned_id;/*!< maximum space id in the existing
293           tables, or assigned during the time
294           mysqld has been up; at an InnoDB
295           startup we scan the data dictionary
296           and set here the maximum of the
297           space id's of the tables there */
298   ib_int64_t  tablespace_version;
299           /*!< a counter which is incremented for
300           every space object memory creation;
301           every space mem object gets a
302           'timestamp' from this; in DISCARD/
303           IMPORT this is used to check if we
304           should ignore an insert buffer merge
305           request */
306   UT_LIST_BASE_NODE_T(fil_space_t) space_list;
307           /*!< list of all file spaces */
308   ibool   space_id_reuse_warned;
309           /* !< TRUE if fil_space_create()
310           has issued a warning about
311           potential space_id reuse */
312 };
```

file system init

- `fil/fil0fil.cc:1676, fil_init()`

```
1672 /*********************************************************************//**
1673 Initializes the tablespace memory cache. */
1674 UNIV_INTERN
1675 void
1676 fil_init(
1677 /*=====*/
1678   ulint hash_size,  /*!< in: hash table size */
1679   ulint max_n_open) /*!< in: max number of open files */
1680 {
1681   ut_a(fil_system == NULL);
1682
1683   ut_a(hash_size > 0);
1684   ut_a(max_n_open > 0);
1685
1686   fil_system = static_cast<fil_system_t*>(
1687     mem_zalloc(sizeof(fil_system_t)));
1688
1689   mutex_create(fil_system_mutex_key,
1690         &fil_system->mutex, SYNC_ANY_LATCH);
1691
1692   fil_system->spaces = hash_create(hash_size);
1693   fil_system->name_hash = hash_create(hash_size);
1694
1695   UT_LIST_INIT(fil_system->LRU);
1696
1697   fil_system->max_n_open = max_n_open;
1698 }
```

Protected by mutex

file system hash

Open file LRU

Tablespace structure

- fil_space_t

```
184 struct fil_space_t {
185   char*    name; /*!< space name = the path to the first file in
186         it */
187   ulint    id; /*!< space id */
188   ib_int64_t  tablespace_version;
189         /*!< in DISCARD/IMPORT this timestamp
190         is used to check if we should ignore
191         an insert buffer merge request for a
192         page because it actually was for the
193         previous incarnation of the space */
194   ibool    mark; /*!< this is set to TRUE at database startup if
195         the space corresponds to a table in the InnoDB
196         data dictionary; so we can print a warning of
197         orphaned tablespaces */
198   ibool    stop_ios;/*!< TRUE if we want to rename the
199         .ibd file of tablespace and want to
200         stop temporarily posting of new i/o
201         requests on the file */
202   ibool    stop_new_ops;
203         /*!< we set this TRUE when we start
204         deleting a single-table tablespace.
205         When this is set following new ops
206         are not allowed:
207         * read IO request
208         * ibuf merge
209         * file flush
210         Note that we can still possibly have
211         new write operations because we don't
212         check this flag when doing flush
213         batches. */
```

Tablespace structure

- fil_space_t

```
214   ulint   purpose;/*!< FIL_TABLESPACE, FIL_LOG, or
215         FIL_ARCH_LOG */
216   UT_LIST_BASE_NODE_T(fil_node_t) chain;
217         /*!< base node for the file chain */
218   ulint   size; /*!< space size in pages; 0 if a single-table
219         tablespace whose size we do not know yet;
220         last incomplete megabytes in data files may be
221         ignored if space == 0 */
222   ulint   flags;  /*!< tablespace flags; see
223         fsp_flags_is_valid(),
224         fsp_flags_get_zip_size() */
225   ulint   n_reserved_extents;
226         /*!< number of reserved free extents for
227         ongoing operations like B-tree page split */
228   ulint   n_pending_flushes; /*!< this is positive when flushing
229         the tablespace to disk; dropping of the
230         tablespace is forbidden if this is positive */
231   ulint   n_pending_ops;/*!< this is positive when we
232         have pending operations against this
233         tablespace. The pending operations can
234         be ibuf merges or lock validation code
235         trying to read a block.
236         Dropping of the tablespace is forbidden
237         if this is positive */
238   hash_node_t hash; /*!< hash chain node */
239   hash_node_t name_hash;/*!< hash chain the name_hash table */
```

Tablespace structure

- fil_space_t

```
240 #ifndef UNIV_HOTBACKUP
241   rw_lock_t latch;  /*!< latch protecting the file space storage
242         allocation */
243 #endif /* !UNIV_HOTBACKUP */
244   UT_LIST_NODE_T(fil_space_t) unflushed_spaces;
245         /*!< list of spaces with at least one unflushed
246         file we have written to */
247   bool    is_in_unflushed_spaces;
248         /*!< true if this space is currently in
249         unflushed_spaces */
250   UT_LIST_NODE_T(fil_space_t) space_list;
251         /*!< list of all spaces */
252   ulint   magic_n;/*!< FIL_SPACE_MAGIC_N */
253 };
```

File node structure

- fil_node_t

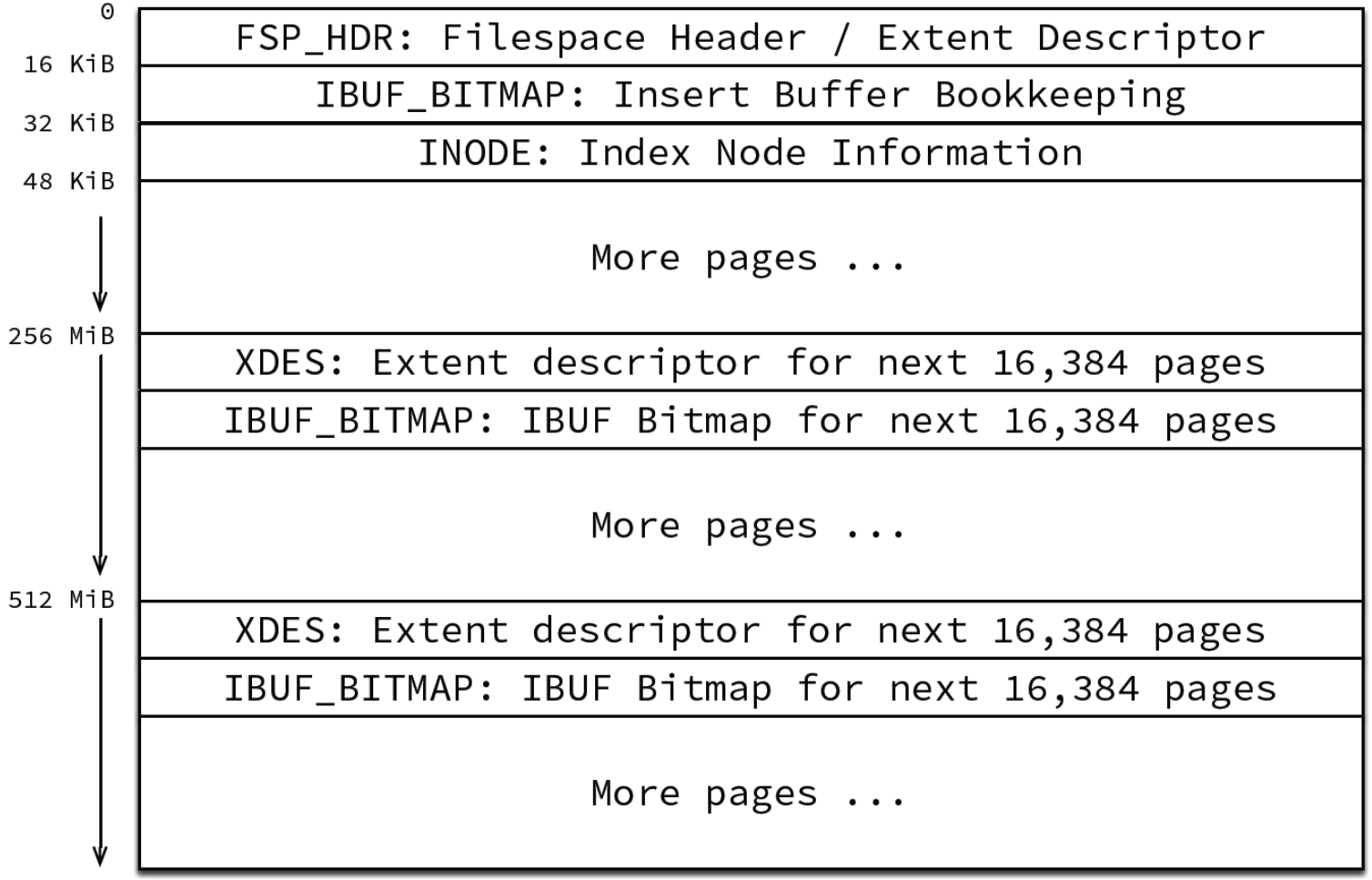
```
144 struct fil_node_t {
145   fil_space_t*  space;  /*!< backpointer to the space
146         where this node belongs */
147   char*    name; /*!< path to the file */
148   ibool    open; /*!< TRUE if file open */
149   os_file_t handle; /*!< OS handle to the file,
150                     if file open */
151   os_event_t  sync_event;/*!< Condition event
152         to group and serialize calls to fsync */
153   ibool    is_raw_disk;/*!< TRUE if the 'file'
154         is actually a raw device or a raw disk partition */
155   ulint    size; /*!< size of the file in database pages,
156         0 if  not known yet; the possible last incomplete
157         megabyte may be ignored if space == 0 */
158   ulint    n_pending;
159         /*!< count of pending i/o's on this file;
160         closing of the file is not allowed if
161         this is > 0 */
```

```
162   ulint    n_pending_flushes;
163         /*!< count of pending flushes on this file;
164         closing of the file is not allowed if
165         this is > 0 */
166   ibool    being_extended;
167         /*!< TRUE if the node is currently
168         being extended. */
169   ib_int64_t  modification_counter;/*!< when we write
170         to the file we increment this by one */
171   ib_int64_t  flush_counter;/*!< up to what
172         modification_counter value we have
173         flushed the modifications to disk */
174   UT_LIST_NODE_T(fil_node_t) chain;
175         /*!< link field for the file chain */
176   UT_LIST_NODE_T(fil_node_t) LRU;
177         /*!< link field for the LRU list */
178   ulint    magic_n;/*!< FIL_NODE_MAGIC_N */
179 };
```

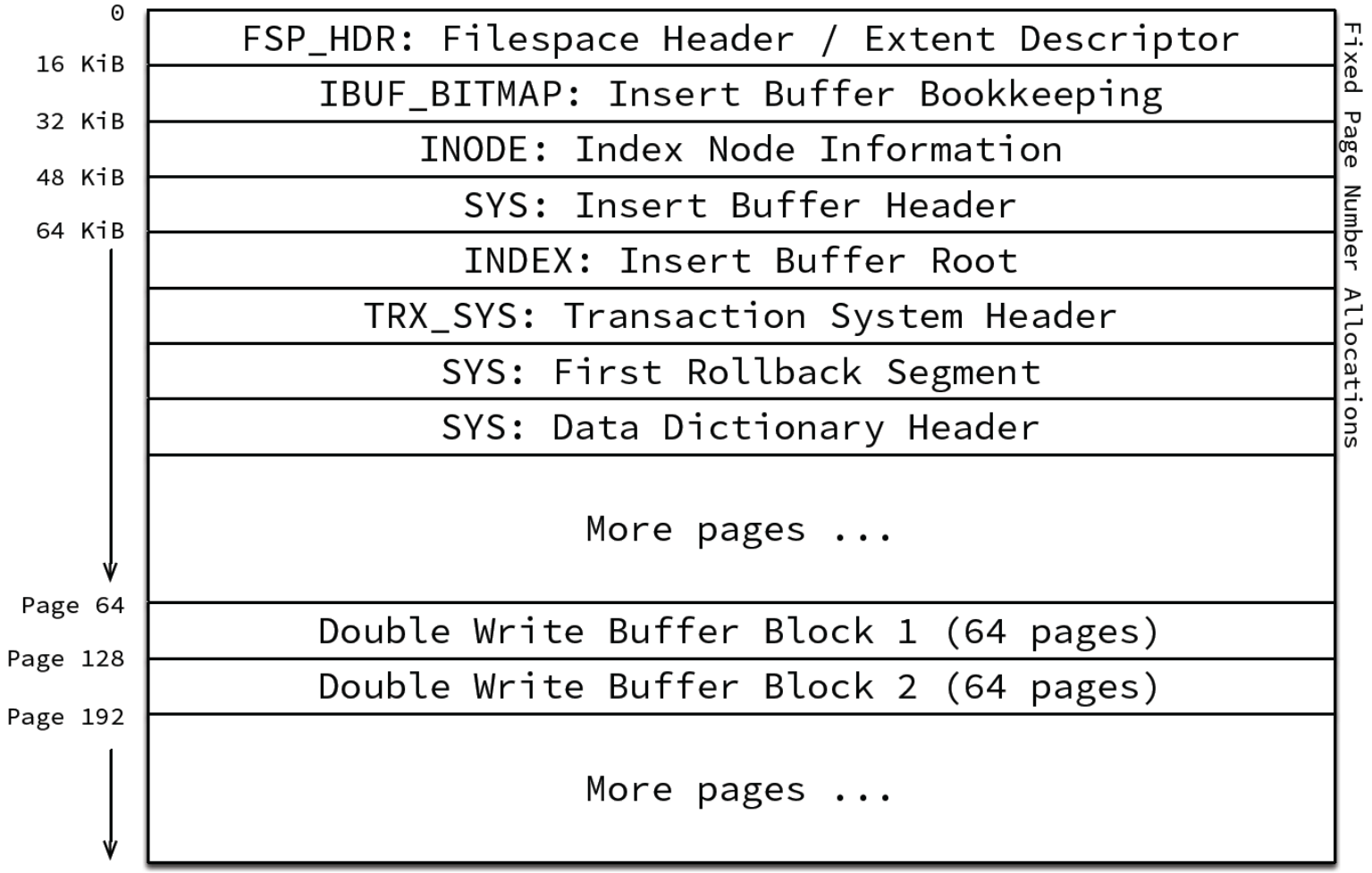
fil_node is file

Note : Space File Structure

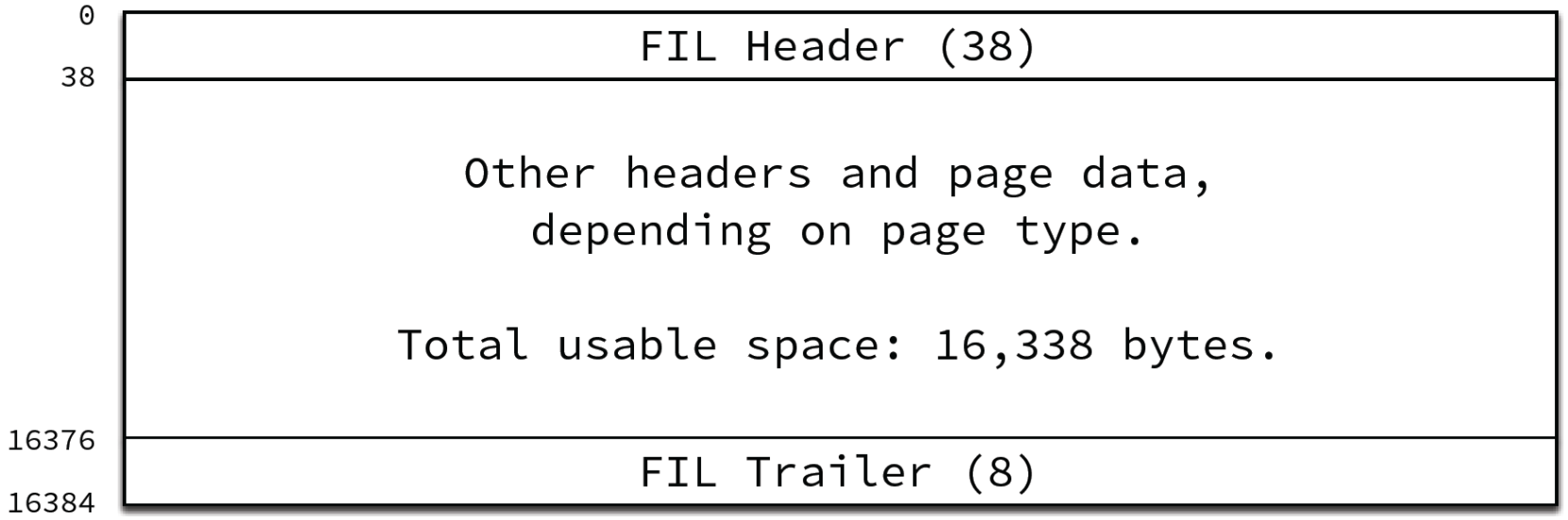
-



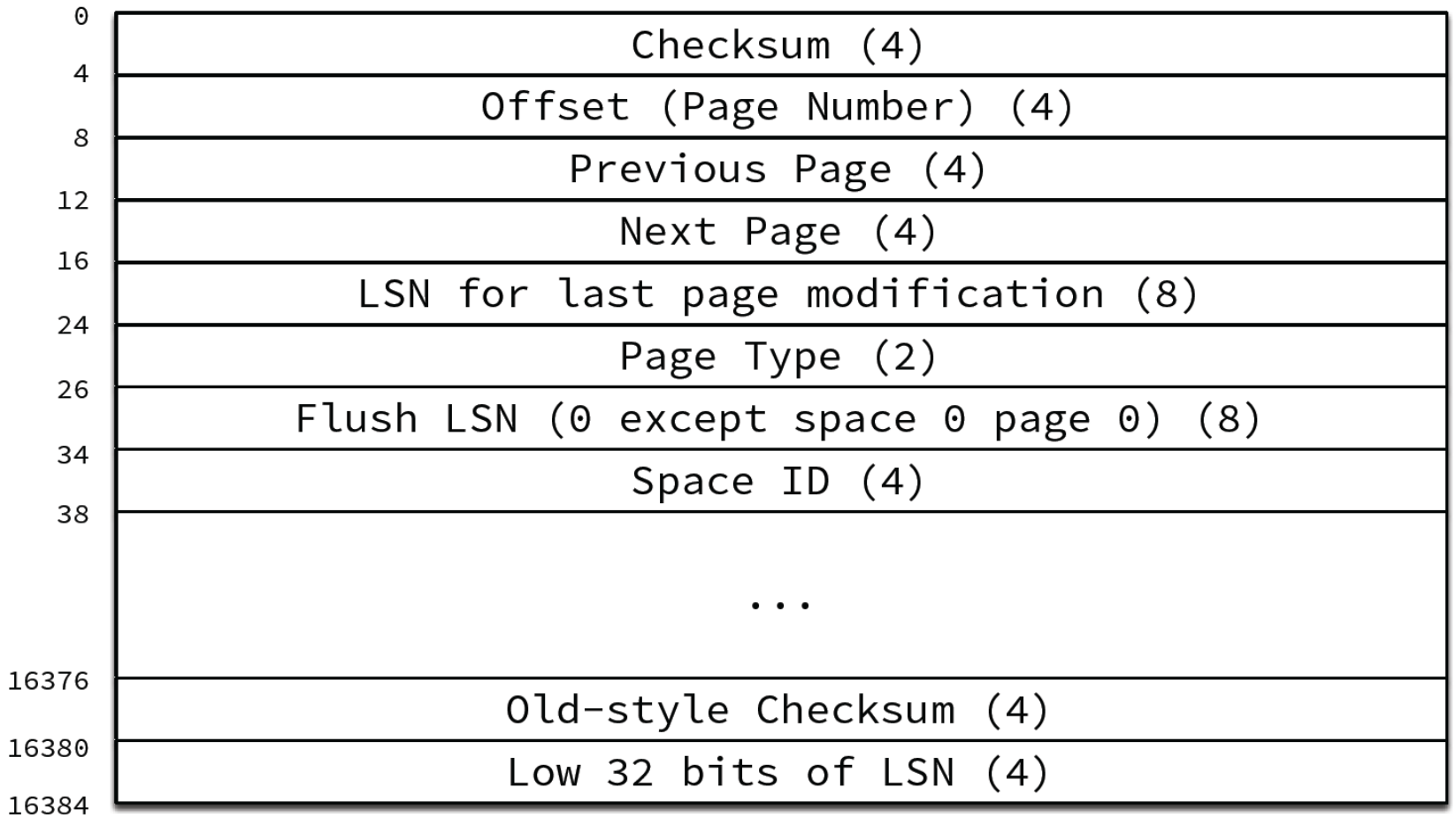
Note : SYSTEM (TRX_SYS) SPACE



Note : Basic Page Structure



Note : FIL Header



issue io operation

- fil_io
  - prepare IO
    - check space id and its status
  - get file node
    - find file node through file node chain in tablespace
  - open file node
  - Do (a)io
  - if synchronous request
    - call fil_node_complete_io
  - return success

issue io operation

- `fil/fil0fil.cc:5466, fil_io()`

```
5466 fil_io(
5467 /*===*/
5468   ulint type,   /*!< in: OS_FILE_READ or OS_FILE_WRITE,
5469         ORed to OS_FILE_LOG, if a log i/o
5470         and ORed to OS_AIO_SIMULATED_WAKE_LATER
5471         if simulated aio and we want to post a
5472         batch of i/os; NOTE that a simulated batch
5473         may introduce hidden chances of deadlocks,
5474         because i/os are not actually handled until
5475         all have been posted: use with great
5476         caution! */
5477   bool  sync,   /*!< in: true if synchronous aio is desired */
5478   ulint space_id, /*!< in: space id */
5479   ulint zip_size, /*!< in: compressed page size in bytes;
5480         0 for uncompressed pages */
5481   ulint block_offset, /*!< in: offset in number of blocks */
5482   ulint byte_offset,  /*!< in: remainder of offset in bytes; in
5483         aio this must be divisible by the OS block
5484         size */
5485   ulint len,    /*!< in: how many bytes to read or write; this
5486         must not cross a file boundary; in aio this
5487         must be a block size multiple */
5488   void* buf,    /*!< in/out: buffer where to store read data
5489         or from where to write; in aio this must be
5490         appropriately aligned */
5491   void* message)  /*!< in: message for aio handler if non-sync
5492         aio used, else ignored */
5493 {
```

issue io operation

- `fil/fil0fil.cc:5466, fil_io()`

```
5493 {
5494   ulint   mode;
5495   fil_space_t*  space;
5496   fil_node_t* node;
5497   ibool   ret;
5498   ulint   is_log;
5499   ulint   wake_later;
5500   os_offset_t offset;
5501   ibool   ignore_nonexistent_pages;
5502
5503   is_log = type & OS_FILE_LOG;
5504   type = type & ~OS_FILE_LOG;
5505
5506   wake_later = type & OS_AIO_SIMULATED_WAKE_LATER;
5507   type = type & ~OS_AIO_SIMULATED_WAKE_LATER;
5508
5509   ignore_nonexistent_pages = type & BUF_READ_IGNORE_NONEXISTENT_PAGES;
5510   type &= ~BUF_READ_IGNORE_NONEXISTENT_PAGES;
5511
5512   ut_ad(byte_offset < UNIV_PAGE_SIZE);
5513   ut_ad(!zip_size || !byte_offset);
5514   ut_ad(ut_is_2pow(zip_size));
5515   ut_ad(buf);
5516   ut_ad(len > 0);
5517   ut_ad(UNIV_PAGE_SIZE == (ulong)(1 << UNIV_PAGE_SIZE_SHIFT));
```

issue io operation

- `fil/fil0fil.cc:5466, fil_io()`

```
5534   if (sync) {
5535     mode = OS_AIO_SYNC;
5536   } else if (is_log) {
5537     mode = OS_AIO_LOG;
5538   } else if (type == OS_FILE_READ
5539        && !recv_no_ibuf_operations
5540        && ibuf_page(space_id, zip_size, block_offset, NULL)) {
5541     mode = OS_AIO_IBUF;
5542   } else {
5543     mode = OS_AIO_NORMAL;
5544   }
```

check mode

issue io operation

- `fil/fil0fil.cc:5466, fil_io()`

```
5550  if (type == OS_FILE_READ) {
5551    srv_stats.data_read.add(len);
5552  } else if (type == OS_FILE_WRITE) {
5553    ut_ad(!srv_read_only_mode);
5554    srv_stats.data_written.add(len);
5555  }
5556
5557  /* Reserve the fil_system mutex and make sure that we can open at
5558  least one file while holding it, if the file is not already open */
5559
5560  fil_mutex_enter_and_prepare_for_io(space_id);
5561
5562  space = fil_space_get_by_id(space_id);
5563
5564  /* If we are deleting a tablespace we don't allow any read
5565  operations on that. However, we do allow write operations. */
5566  if (space == 0 || (type == OS_FILE_READ && space->stop_new_ops)) {
5567    mutex_exit(&fil_system->mutex);
5568
5569    ib_logf(IB_LOG_LEVEL_ERROR,
5570      "Trying to do i/o to a tablespace which does "
5571      "not exist. i/o type %lu, space id %lu, "
5572      "page no. %lu, i/o length %lu bytes",
5573      (ulong) type, (ulong) space_id, (ulong) block_offset,
5574      (ulong) len);
5575
5576    return(DB_TABLESPACE_DELETED);
5577  }
```

enter mutex and prepare io for current space

get tablespace using hash table

Missing or IO not allowed error handling

issue io operation - prepare for io

- `fil/fil0fil.cc:1007, fil_mutex_enter_and_prepare_for_io()`

```
1006 void
1007 fil_mutex_enter_and_prepare_for_io(
1008 /*=================================*/
1009   ulint space_id) /*!< in: space id */
1010 {
1011   fil_space_t*  space;
1012   ibool   success;
1013   ibool   print_info  = FALSE;
1014   ulint   count    = 0;
1015   ulint   count2    = 0;
1016
1017 retry:
1018   mutex_enter(&fil_system->mutex);
1019
1020   if (space_id == 0 || space_id >= SRV_LOG_SPACE_FIRST_ID) {
1021     /* We keep log files and system tablespace files always open;
1022     this is important in preventing deadlocks in this module, as
1023     a page read completion often performs another read from the
1024     insert buffer. The insert buffer is in tablespace 0, and we
1025     cannot end up waiting in this function. */
1026
1027     return;
1028   }
1029
1030   space = fil_space_get_by_id(space_id);
```

if current space if SYSTEM or LOG then return

get tablespace using hash table

issue io operation - prepare for io

- `fil/fil0fil.cc:1007, fil_mutex_enter_and_prepare_for_io()`

```
1032   if (space != NULL && space->stop_ios) {
1033     /* We are going to do a rename file and want to stop new i/o's
1034     for a while */
1035
1036     if (count2 > 20000) {
1037       fputs("InnoDB: Warning: tablespace ", stderr);
1038       ut_print_filename(stderr, space->name);
1039       fprintf(stderr,
1040         " has i/o ops stopped for a long time %lu\n",
1041         (ulong) count2);
1042     }
1043
1044     mutex_exit(&fil_system->mutex);
1045
1046 #ifndef UNIV_HOTBACKUP
1047
1048     /* Wake the i/o-handler threads to make sure pending
1049     i/o's are performed */
1050     os_aio_simulated_wake_handler_threads();
1051
1052     /* The sleep here is just to give IO helper threads a
1053     bit of time to do some work. It is not required that
1054     all IO related to the tablespace being renamed must
1055     be flushed here as we do fil_flush() in
1056     fil_rename_tablespace() as well. */
1057     os_thread_sleep(20000);
1058
1059 #endif /* UNIV_HOTBACKUP */
```

```
1060
1061     /* Flush tablespaces so that we can close modified
1062     files in the LRU list */
1063     fil_flush_file_spaces(FIL_TABLESPACE);
1064
1065     os_thread_sleep(20000);
1066
1067     count2++;
1068
1069     goto retry;
1070   }
1071
1072   if (fil_system->n_open < fil_system->max_n_open) {
1073
1074     return;
1075   }
```

in case we can’t do IO right now :
IO, sleep, sync, sleep

issue io operation - prepare for io

- `fil/fil0fil.cc:1007, fil_mutex_enter_and_prepare_for_io()`

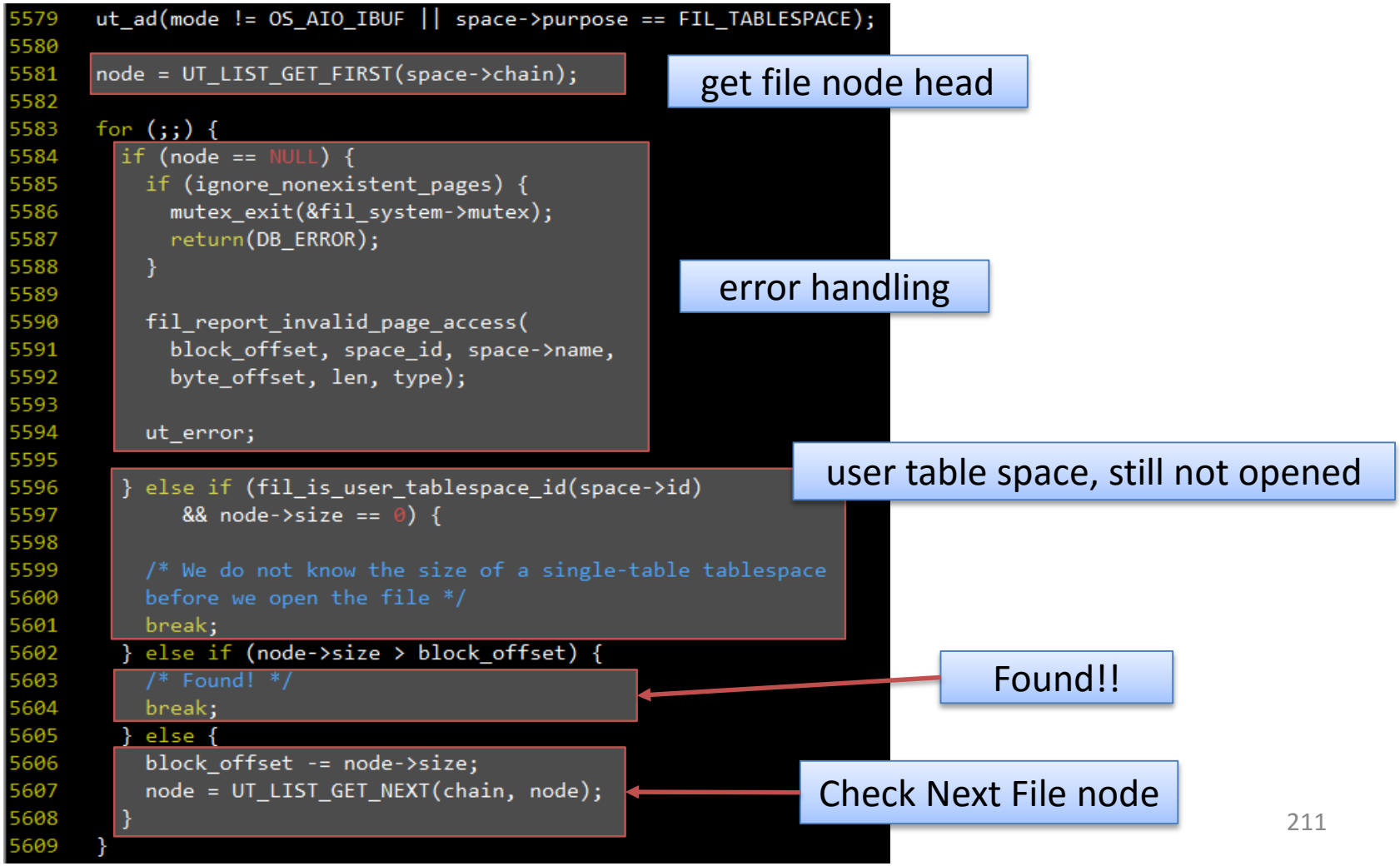
```
1089
1090   /* Too many files are open, try to close some */
1091 close_more:
1092   success = fil_try_to_close_file_in_LRU(print_info);
1093
1094   if (success && fil_system->n_open >= fil_system->max_n_open) {
1095
1096     goto close_more;
1097   }
1098
1099   if (fil_system->n_open < fil_system->max_n_open) {
1100     /* Ok */
1101
1102     return;
1103   }
```

too many file open, close file in LRU manner

return back

issue io operation

- `fil/fil0fil.cc:5466, fil_io()`



issue io operation

- `fil/fil0fil.cc:5466, fil_io()`

```
5611  /* Open file if closed */
5612  if (!fil_node_prepare_for_io(node, fil_system, space)) {
5613    if (space->purpose == FIL_TABLESPACE
5614        && fil_is_user_tablespace_id(space->id)) {
5615      mutex_exit(&fil_system->mutex);
5616
5617      ib_logf(IB_LOG_LEVEL_ERROR,
5618        "Trying to do i/o to a tablespace which "
5619        "exists without .ibd data file. "
5620        "i/o type %lu, space id %lu, page no %lu, "
5621        "i/o length %lu bytes",
5622        (ulong) type, (ulong) space_id,
5623        (ulong) block_offset, (ulong) len);
5624
5625      return(DB_TABLESPACE_DELETED);
5626    }
5627
5628    /* The tablespace is for log. Currently, we just assert here
5629    to prevent handling errors along the way fil_io returns.
5630    Also, if the log files are missing, it would be hard to
5631    promise the server can continue running. */
5632    ut_a(0);
5633  }
```

prepare file node - open a file

issue io operation - prepare fil_node

- `fil/fil0fil.cc:5342, fil_node_prepare_for_io()`

```
5341 bool
5342 fil_node_prepare_for_io(
5343 /*=====================*/
5344   fil_node_t* node, /*!< in: file node */
5345   fil_system_t* system, /*!< in: tablespace memory cache */
5346   fil_space_t*  space)  /*!< in: space */
5347 {
5348   ut_ad(node && system && space);
5349   ut_ad(mutex_own(&(system->mutex)));
5350
5351   if (system->n_open > system->max_n_open + 5) {
5352     ut_print_timestamp(stderr);
5353     fprintf(stderr,
5354       "  InnoDB: Warning: open files %lu"
5355       " exceeds the limit %lu\n",
5356       (ulong) system->n_open,
5357       (ulong) system->max_n_open);
5358   }
5359
5360   if (node->open == FALSE) {
5361     /* File is closed: open it */
5362     ut_a(node->n_pending == 0);
5363
5364     if (!fil_node_open_file(node, system, space)) {
5365       return(false);
5366     }
5367   }
```

node open file

issue io operation - open fil node

- `fil/fil0fil.cc:701, fil_node_open_file ()`

```
700 bool
701 fil_node_open_file(
702 /*===============*/
703   fil_node_t* node, /*!< in: file node */
704   fil_system_t* system, /*!< in: tablespace memory cache */
705   fil_space_t*  space)  /*!< in: space */
706 {
707   os_offset_t size_bytes;
708   ibool   ret;
709   ibool   success;
710   byte*   buf2;
711   byte*   page;
712   ulint   space_id;
713   ulint   flags;
714   ulint   page_size;
715
716   ut_ad(mutex_own(&(system->mutex)));
717   ut_a(node->n_pending == 0);
718   ut_a(node->open == FALSE);
```

issue io operation - open fil node

- `fil/fil0fil.cc:701, fil_node_open_file ()`

```
720   if (node->size == 0) {
721     /* It must be a single-table tablespace and we do not know the
722     size of the file yet. First we open the file in the normal
723     mode, no async I/O here, for simplicity. Then do some checks,
724     and close the file again.
725     NOTE that we could not use the simple file read function
726     os_file_read() in Windows to read from a file opened for
727     async I/O! */
728
729     node->handle = os_file_create_simple_no_error_handling(
730       innodb_file_data_key, node->name, OS_FILE_OPEN,
731       OS_FILE_READ_ONLY, &success);
732     if (!success) {
733       /* The following call prints an error message */
734       os_file_get_last_error(true);
735
736       ut_print_timestamp(stderr);
737
738       ib_logf(IB_LOG_LEVEL_WARN, "InnoDB: Error: cannot "
739         "open %s\n. InnoDB: Have you deleted .ibd "
740         "files under a running mysqld server?\n",
741         node->name);
742
743       return(false);
744     }
```

This function open file

issue io operation - open fil node

- `fil/fil0fil.cc:701, fil_node_open_file ()`

```
746     size_bytes = os_file_get_size(node->handle);
747     ut_a(size_bytes != (os_offset_t) -1);
748 #ifdef UNIV_HOTBACKUP
749     if (space->id == 0) {
750       node->size = (ulint) (size_bytes / UNIV_PAGE_SIZE);
751       os_file_close(node->handle);
752       goto add_size;
753     }
754 #endif /* UNIV_HOTBACKUP */
755     ut_a(space->purpose != FIL_LOG);
756     ut_a(fil_is_user_tablespace_id(space->id));
757
758     if (size_bytes < FIL_IBD_FILE_INITIAL_SIZE * UNIV_PAGE_SIZE) {
759       fprintf(stderr,
760         "InnoDB: Error: the size of single-table"
761         " tablespace file %s\n"
762         "InnoDB: is only " UINT64PF ","
763         " should be at least %lu!\n",
764         node->name,
765         size_bytes,
766         (ulong) (FIL_IBD_FILE_INITIAL_SIZE
767            * UNIV_PAGE_SIZE));
768
769       ut_a(0);
770     }
```

measure file size using lseek (SEEK_END)

file size is too small : error handling

issue io operation - open fil node

- `fil/fil0fil.cc:701, fil_node_open_file ()`

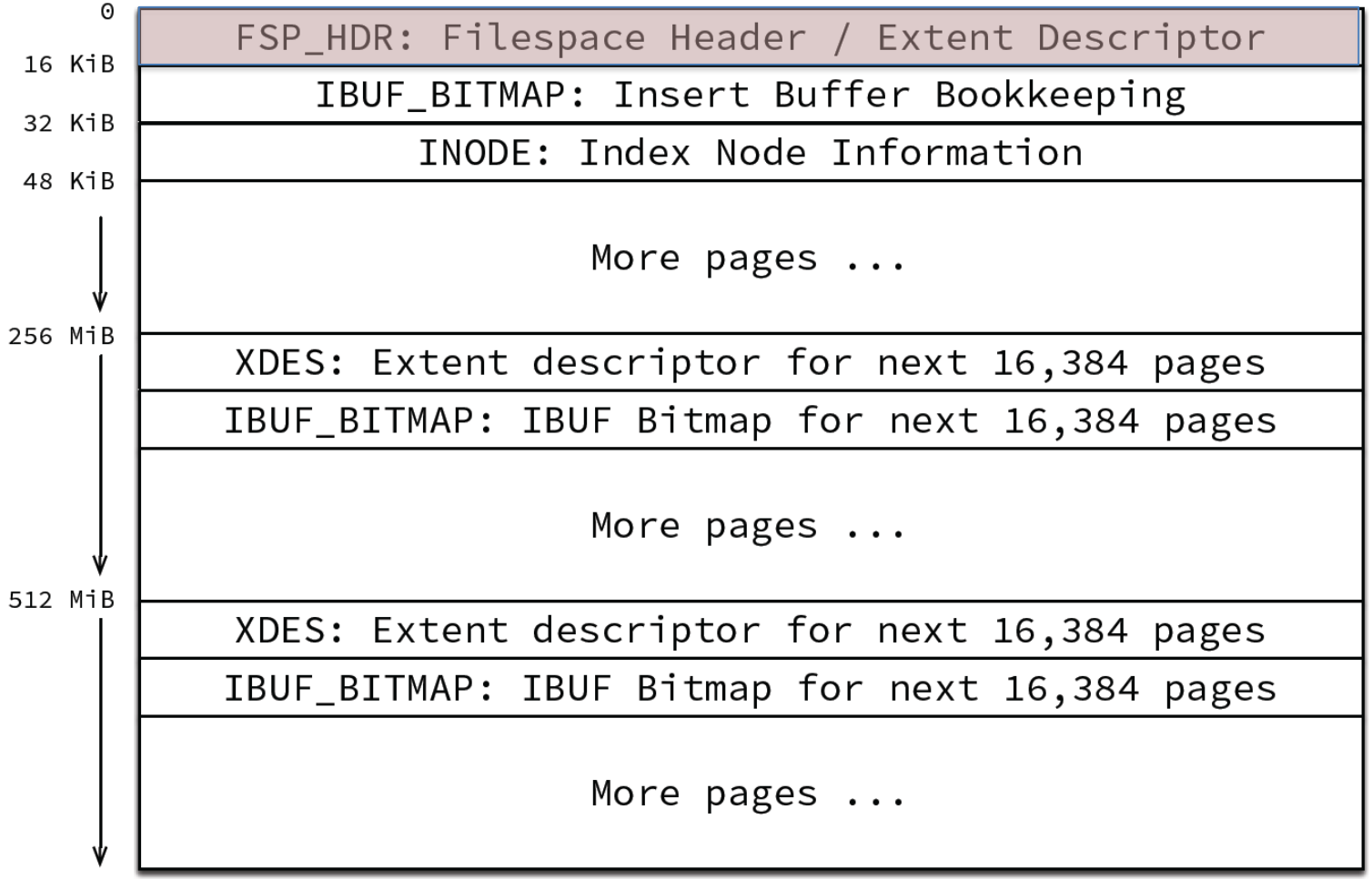
```
772     /* Read the first page of the tablespace */
773
774     buf2 = static_cast<byte*>(ut_malloc(2 * UNIV_PAGE_SIZE));
775     /* Align the memory for file i/o if we might have O_DIRECT
776     set */
777     page = static_cast<byte*>(ut_align(buf2, UNIV_PAGE_SIZE));
778
779     success = os_file_read(node->handle, page, 0, UNIV_PAGE_SIZE);
780     space_id = fsp_header_get_space_id(page);
781     flags = fsp_header_get_flags(page);
782     page_size = fsp_flags_get_page_size(flags);
783
784     ut_free(buf2);
785
786     /* Close the file now that we have read the space id from it */
787
788     os_file_close(node->handle);
789
790     if (UNIV_UNLIKELY(space_id != space->id)) {
791       fprintf(stderr,
792         "InnoDB: Error: tablespace id is %lu"
793         " in the data dictionary\n"
794         "InnoDB: but in file %s it is %lu!\n",
795         space->id, node->name, space_id);
796
797       ut_error;
798     }
```

read the 1st page and get information

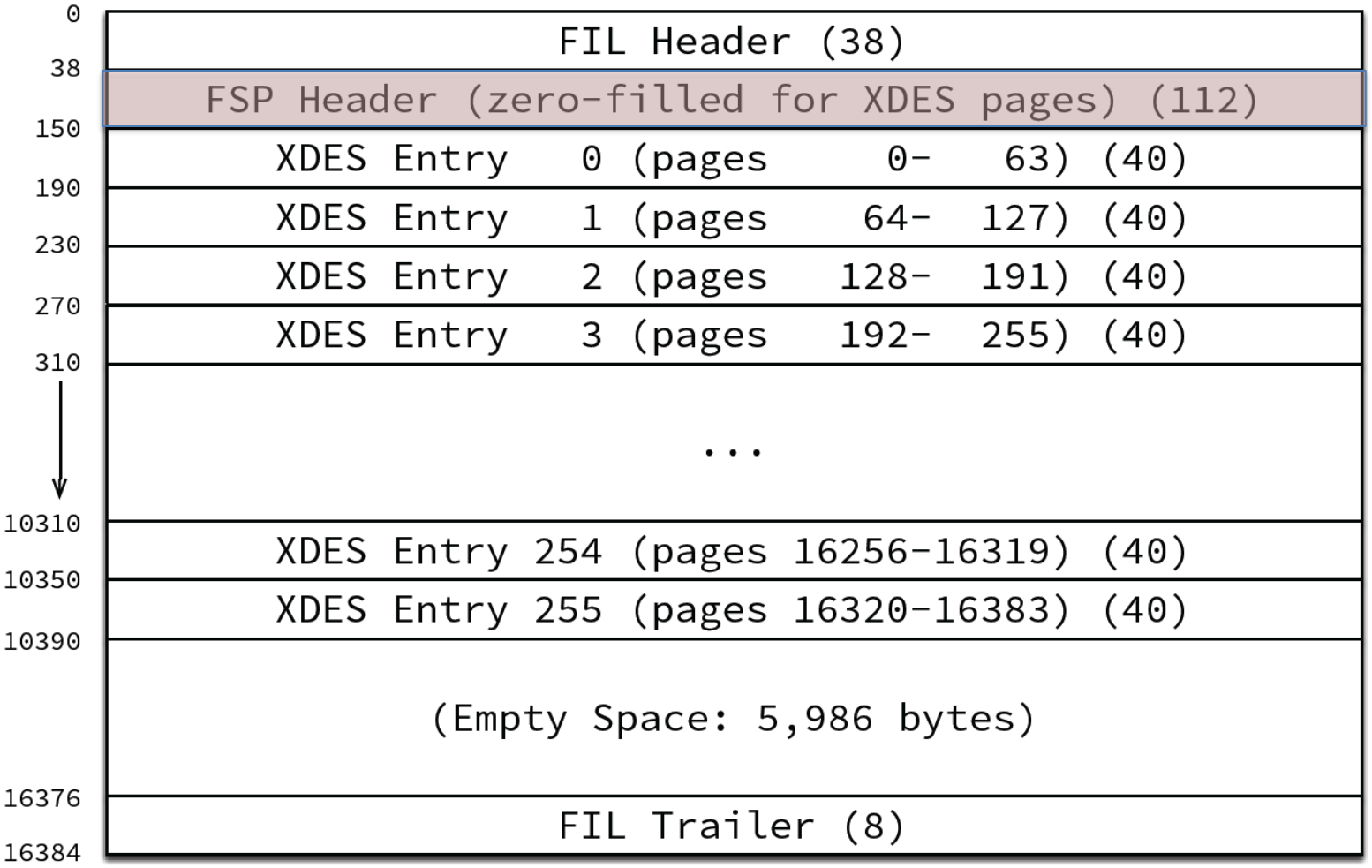
close file

Note : Space File Structure

-

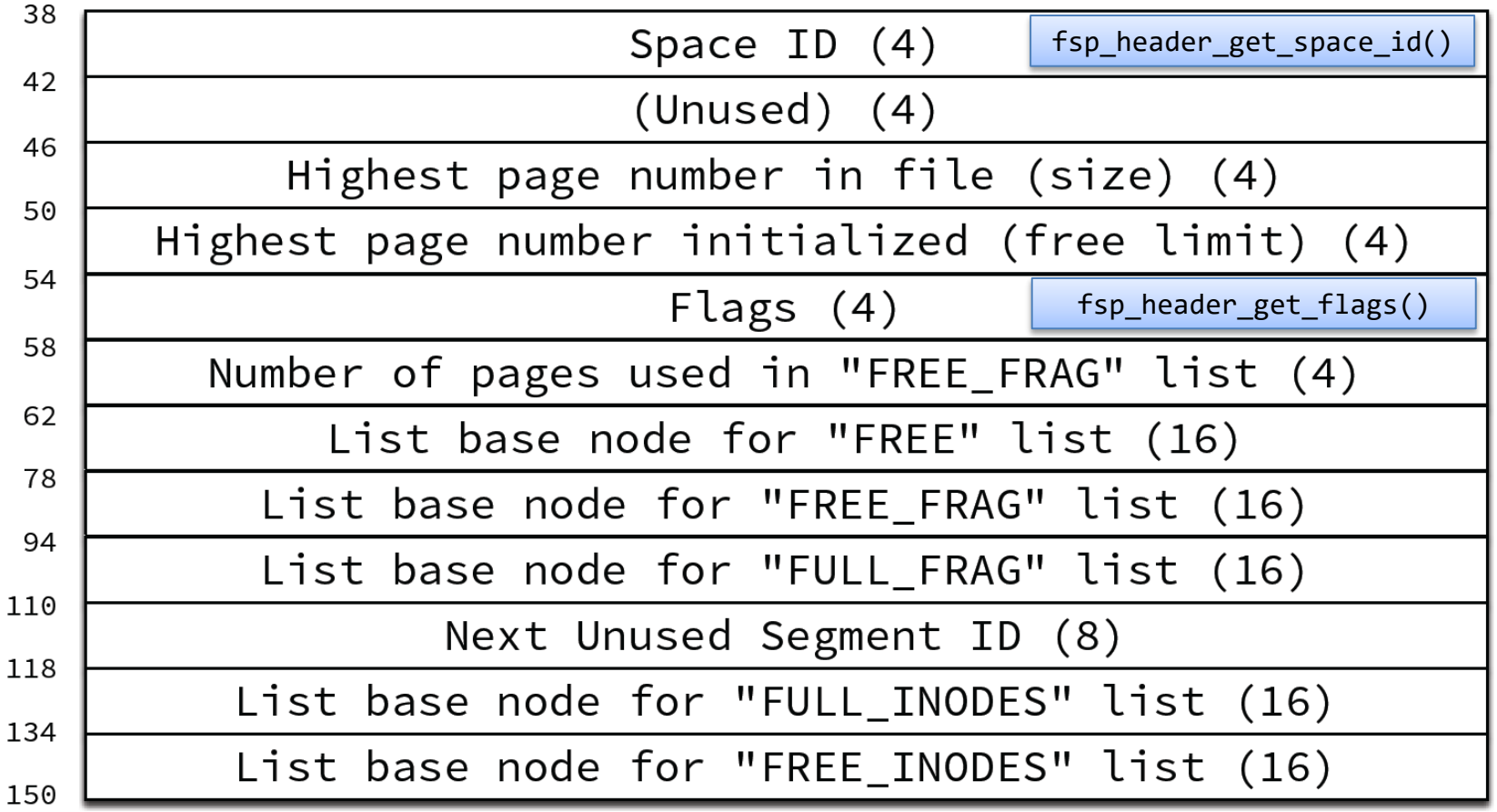


Note : FSP_HDR

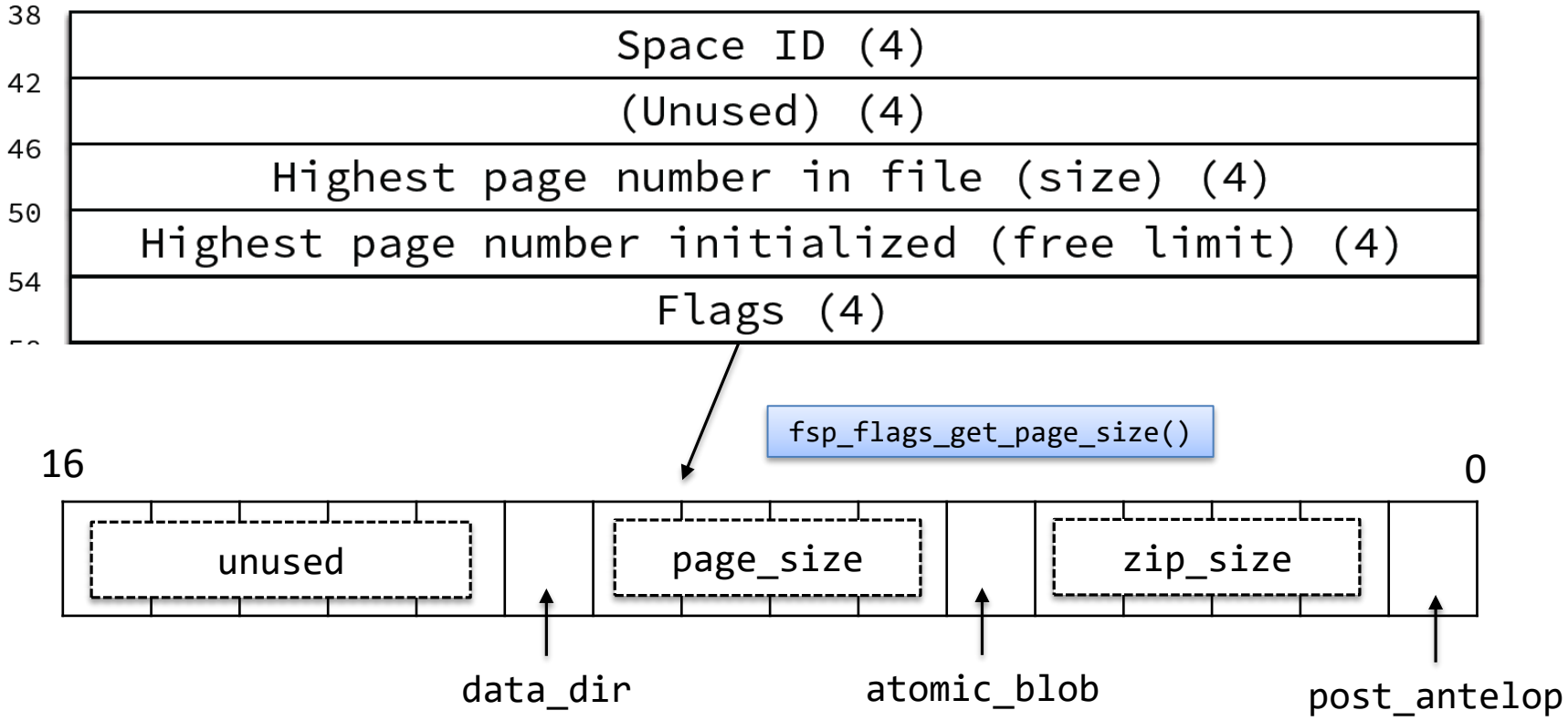

0
FIL Header (38)
38
FSP Header (zero-filled for XDES pages) (112)
150
XDES Entry 0 (pages 0- 63) (40)
190
XDES Entry 1 (pages 64- 127) (40)
230
XDES Entry 2 (pages 128- 191) (40)
270
XDES Entry 3 (pages 192- 255) (40)
310
...
10310
XDES Entry 254 (pages 16256-16319) (40)
10350
XDES Entry 255 (pages 16320-16383) (40)
10390
(Empty Space: 5,986 bytes)
16376
FIL Trailer (8)
16384

Note : FSP_HDR

This is what we just read of



Note : FSP_HDR flag get page size



issue io operation - open fil node

- `fil/fil0fil.cc:701, fil_node_open_file ()`

```
790     if (UNIV_UNLIKELY(space_id != space->id)) {
791       fprintf(stderr,
792         "InnoDB: Error: tablespace id is %lu"
793         " in the data dictionary\n"
794         "InnoDB: but in file %s it is %lu!\n",
795         space->id, node->name, space_id);
796
797       ut_error;
798     }
799
800     if (UNIV_UNLIKELY(space_id == ULINT_UNDEFINED
801           || space_id == 0)) {
802       fprintf(stderr,
803         "InnoDB: Error: tablespace id %lu"
804         " in file %s is not sensible\n",
805         (ulong) space_id, node->name);
806
807       ut_error;
808     }
```

```
810     if (UNIV_UNLIKELY(fsp_flags_get_page_size(space->flags)
811           != page_size)) {
812       fprintf(stderr,
813         "InnoDB: Error: tablespace file %s"
814         " has page size 0x%lx\n"
815         "InnoDB: but the data dictionary"
816         " expects page size 0x%lx!\n",
817         node->name, flags,
818         fsp_flags_get_page_size(space->flags));
819
820       ut_error;
821     }
822
823     if (UNIV_UNLIKELY(space->flags != flags)) {
824       fprintf(stderr,
825         "InnoDB: Error: table flags are 0x%lx"
826         " in the data dictionary\n"
827         "InnoDB: but the flags in file %s are 0x%lx!\n",
828         space->flags, node->name, flags);
829
830       ut_error;
831     }
```

check flag info is valid

issue io operation - open fil node

- `fil/fil0fil.cc:701, fil_node_open_file ()`

```
833     if (size_bytes >= 1024 * 1024) {
834       /* Truncate the size to whole megabytes. */
835       size_bytes = ut_2pow_round(size_bytes, 1024 * 1024);
836     }
837
838     if (!fsp_flags_is_compressed(flags)) {
839       node->size = (ulint) (size_bytes / UNIV_PAGE_SIZE);
840     } else {
841       node->size = (ulint)
842         (size_bytes
843          / fsp_flags_get_zip_size(flags));
844     }
845
846 #ifdef UNIV_HOTBACKUP
847 add_size:
848 #endif /* UNIV_HOTBACKUP */
849     space->size += node->size;
850   }
```

Finally, we can set size of current node

issue io operation - open fil node

- `fil/fil0fil.cc:701, fil_node_open_file ()`

```
858   if (space->purpose == FIL_LOG) {
859     node->handle = os_file_create(innodb_file_log_key,
860                   node->name, OS_FILE_OPEN,
861                   OS_FILE_AIO, OS_LOG_FILE,
862                   &ret);
863   } else if (node->is_raw_disk) {
864     node->handle = os_file_create(innodb_file_data_key,
865                   node->name,
866                   OS_FILE_OPEN_RAW,
867                   OS_FILE_AIO, OS_DATA_FILE,
868                    &ret);
869   } else {
870     node->handle = os_file_create(innodb_file_data_key,
871                   node->name, OS_FILE_OPEN,
872                   OS_FILE_AIO, OS_DATA_FILE,
873                   &ret);
874   }
875
876   ut_a(ret);
877
878   node->open = TRUE;
879
880   system->n_open++;
881   fil_n_file_opened++;
882
883   if (fil_space_belongs_in_lru(space)) {
884
885     /* Put the node to the LRU list */
886     UT_LIST_ADD_FIRST(LRU, system->LRU, node);
887   }
888
889   return(true);
890 }
```

open file, **see this later**

put current node to LRU list

step out

issue io operation - prepare fil_node

- `fil/fil0fil.cc:5342, fil_node_prepare_for_io()`

```
5370  if (node->n_pending == 0 && fil_space_belongs_in_lru(space)) {
5371    /* The node is in the LRU list, remove it */
5372
5373    ut_a(UT_LIST_GET_LEN(system->LRU) > 0);
5374
5375    UT_LIST_REMOVE(LRU, system->LRU, node);
5376  }
5377
5378  node->n_pending++;
5379
5380  return(true);
5381 }
```

remove current node from LRU list

Why remove ??

in this LRU, add a node after IO complete

step out

issue io operation

- `fil/fil0fil.cc:5466, fil_io()`

```
5611  /* Open file if closed */
5612  if (!fil_node_prepare_for_io(node, fil_system, space)) {
5613    if (space->purpose == FIL_TABLESPACE
5614        && fil_is_user_tablespace_id(space->id)) {
5615      mutex_exit(&fil_system->mutex);
5616
5617      ib_logf(IB_LOG_LEVEL_ERROR,
5618        "Trying to do i/o to a tablespace which "
5619        "exists without .ibd data file. "
5620        "i/o type %lu, space id %lu, page no %lu, "
5621        "i/o length %lu bytes",
5622        (ulong) type, (ulong) space_id,
5623        (ulong) block_offset, (ulong) len);
5624
5625      return(DB_TABLESPACE_DELETED);
5626    }
5627
5628    /* The tablespace is for log. Currently, we just assert here
5629    to prevent handling errors along the way fil_io returns.
5630    Also, if the log files are missing, it would be hard to
5631    promise the server can continue running. */
5632    ut_a(0);
5633  }
```

prepare file node - open a file

issue io operation

- `fil/fil0fil.cc:5466, fil_io()`

```
5636  /* Check that at least the start offset is within the bounds of a
5637  single-table tablespace, including rollback tablespaces. */
5638  if (UNIV_UNLIKELY(node->size <= block_offset)
5639      && space->id != 0 && space->purpose == FIL_TABLESPACE) {
5640
5641    fil_report_invalid_page_access(
5642      block_offset, space_id, space->name, byte_offset,
5643      len, type);
5644
5645    ut_error;
5646  }
5647
5648  /* Now we have made the changes in the data structures of fil_system */
5649  mutex_exit(&fil_system->mutex);
5650
```

issue io operation

- `fil/fil0fil.cc:5466, fil_io()`

```
5651  /* Calculate the low 32 bits and the high 32 bits of the file offset */
5652
5653  if (!zip_size) {
5654    offset = ((os_offset_t) block_offset << UNIV_PAGE_SIZE_SHIFT)
5655      + byte_offset;
5656
5657    ut_a(node->size - block_offset
5658         >= ((byte_offset + len + (UNIV_PAGE_SIZE - 1))
5659       / UNIV_PAGE_SIZE));
5660  } else {
5661    ulint zip_size_shift;
5662    switch (zip_size) {
5663    case 1024: zip_size_shift = 10; break;
5664    case 2048: zip_size_shift = 11; break;
5665    case 4096: zip_size_shift = 12; break;
5666    case 8192: zip_size_shift = 13; break;
5667    case 16384: zip_size_shift = 14; break;
5668    default: ut_error;
5669    }
5670    offset = ((os_offset_t) block_offset << zip_size_shift)
5671      + byte_offset;
5672    ut_a(node->size - block_offset
5673         >= (len + (zip_size - 1)) / zip_size);
5674  }
```

calculate byte offset

issue io operation

- `fil/fil0fil.cc:5466, fil_io()`

```
5676   /* Do aio */
5677
5678   ut_a(byte_offset % OS_FILE_LOG_BLOCK_SIZE == 0);
5679   ut_a((len % OS_FILE_LOG_BLOCK_SIZE) == 0);
5680
5691   /* Queue the aio request */
5692   ret = os_aio(type, mode | wake_later, node->name, node->handle, buf,
5693          offset, len, node, message);
5695   ut_a(ret);
5696
5697   if (mode == OS_AIO_SYNC) {
5698     /* The i/o operation is already completed when we return from
5699     os_aio: */
5700
5701     mutex_enter(&fil_system->mutex);
5702
5703     fil_node_complete_io(node, fil_system, type);
5704
5705     mutex_exit(&fil_system->mutex);
5706
5707     ut_ad(fil_validate_skip());
5708   }
5709
5710   return(DB_SUCCESS);
5711 }
```

see this later.

file node comple io

issue io operation

- `fil/fil0fil.cc:5388, fil_node_complete_io()`

```
5387 void
5388 fil_node_complete_io(
5389 /*===================*/
5390   fil_node_t* node, /*!< in: file node */
5391   fil_system_t* system, /*!< in: tablespace memory cache */
5392   ulint   type) /*!< in: OS_FILE_WRITE or OS_FILE_READ; marks
5393         the node as modified if
5394         type == OS_FILE_WRITE */
5395 {
5396   ut_ad(node);
5397   ut_ad(system);
5398   ut_ad(mutex_own(&(system->mutex)));
5399
5400   ut_a(node->n_pending > 0);
5401
5402   node->n_pending--;
5403
5404   if (type == OS_FILE_WRITE) {
5405     ut_ad(!srv_read_only_mode);
5406     system->modification_counter++;
5407     node->modification_counter = system->modification_counter;
5408
5409     if (fil_buffering_disabled(node->space)) {
5410
5411       /* We don't need to keep track of unflushed
5412       changes as user has explicitly disabled
5413       buffering. */
5414       ut_ad(!node->space->is_in_unflushed_spaces);
5415       node->flush_counter = node->modification_counter;
5416
```

decrease # of pending io

change system modification counter -> this affect background flushing

change node modification counter -> this affect fil_flush (fsync or not)

issue io operation

- `fil/fil0fil.cc:5388, fil_node_complete_io()`

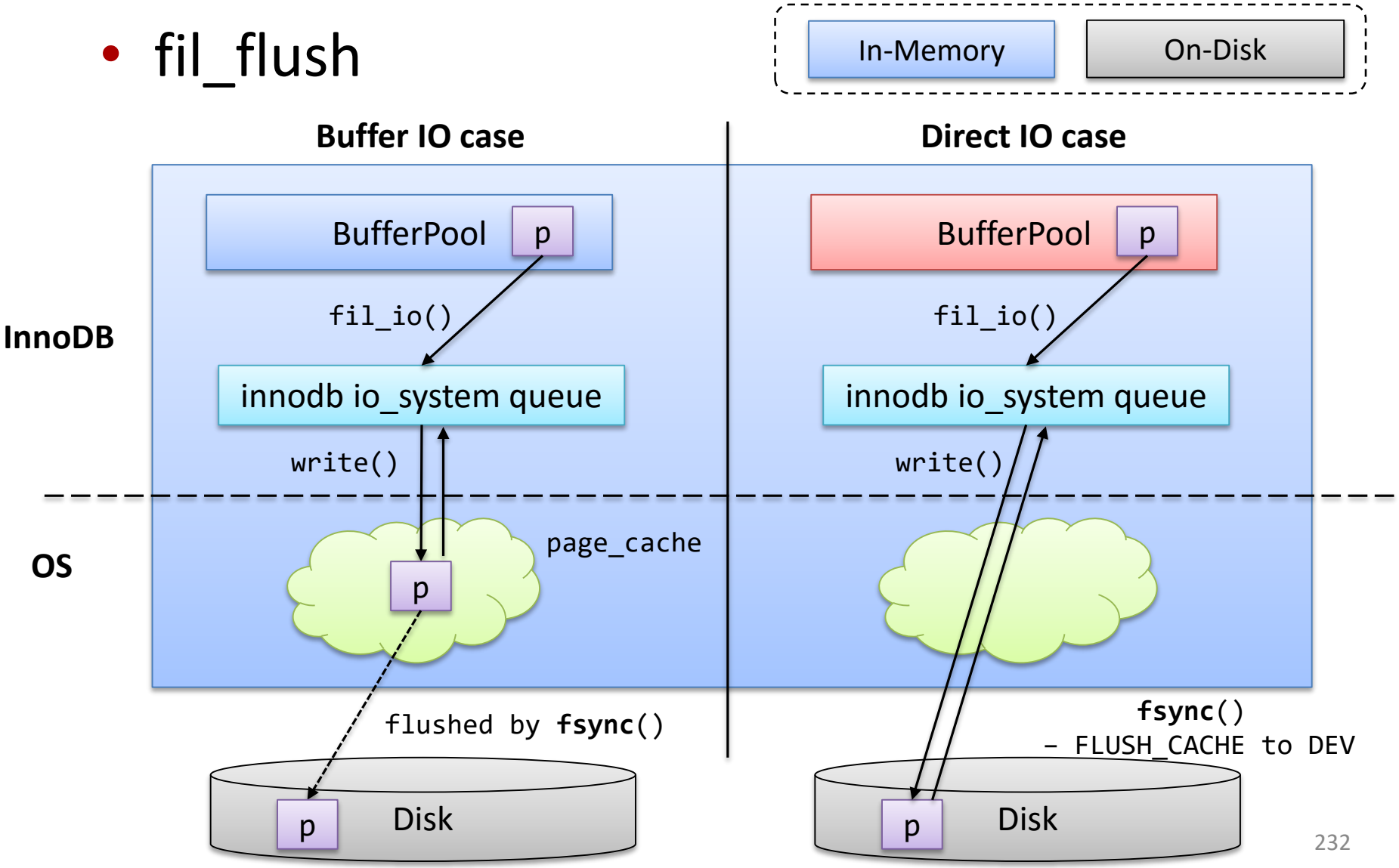
```
5417      } else if (!node->space->is_in_unflushed_spaces) {
5418
5419        node->space->is_in_unflushed_spaces = true;
5420        UT_LIST_ADD_FIRST(unflushed_spaces,
5421              system->unflushed_spaces,
5422              node->space);
5423      }
5424    }
5425
5426    if (node->n_pending == 0 && fil_space_belongs_in_lru(node->space)) {
5427
5428      /* The node must be put back to the LRU list */
5429      UT_LIST_ADD_FIRST(LRU, system->LRU, node);
5430    }
5431  }
```

add current space to unflushed_spaces

if current node not added to LRU list yet, then add this node

sync buffered write to disk

- fil_flush



sync buffered write to disk

- `fil/fil0fil.cc:5789, fil_flush()`

```
5788 void
5789 fil_flush(
5790 /*======*/
5791   ulint space_id) /*!< in: file space id (this can be a group of
5792         log files or a tablespace of the database) */
5793 {
5794   fil_space_t*  space;
5795   fil_node_t* node;
5796   os_file_t file;
5797
5798
5799   mutex_enter(&fil_system->mutex);
5800
5801   space = fil_space_get_by_id(space_id);
5802
5803   if (!space || space->stop_new_ops) {
5804     mutex_exit(&fil_system->mutex);
5805
5806     return;
5807   }
```

sync buffered write to disk

- `fil/fil0fil.cc:5789, fil_flush()`

```
5809    if (fil_buffering_disabled(space)) {
5810
5811      /* No need to flush. User has explicitly disabled
5812      buffering. */
5813      ut_ad(!space->is_in_unflushed_spaces);
5814      ut_ad(fil_space_is_flushed(space));
5815      ut_ad(space->n_pending_flushes == 0);
5816
5817 #ifdef UNIV_DEBUG
5818      for (node = UT_LIST_GET_FIRST(space->chain);
5819           node != NULL;
5820           node = UT_LIST_GET_NEXT(chain, node)) {
5821        ut_ad(node->modification_counter
5822              == node->flush_counter);
5823        ut_ad(node->n_pending_flushes == 0);
5824      }
5825 #endif /* UNIV_DEBUG */
5826
5827      mutex_exit(&fil_system->mutex);
5828      return;
5829    }
```

flush_method = NO_FSYNC case, do nothing for this function

```
322 /** Determine if user has explicitly disabled fsync(). */
323 #ifndef __WIN__
324 # define fil_buffering_disabled(s)   \
325   ((s)->purpose == FIL_TABLESPACE \
326    && srv_unix_file_flush_method  \
327    == SRV_UNIX_O_DIRECT_NO_FSYNC)
```

sync buffered write to disk

- `fil/fil0fil.cc:5789, fil_flush()`

```
5831  space->n_pending_flushes++; /*!< prevent dropping of the space while
5832          we are flushing */
5833  for (node = UT_LIST_GET_FIRST(space->chain);
5834       node != NULL;
5835       node = UT_LIST_GET_NEXT(chain, node)) {
5836
5837    ib_int64_t old_mod_counter = node->modification_counter;;
5838
5839    if (old_mod_counter <= node->flush_counter) {
5840      continue;
5841    }
5842
5843    ut_a(node->open);
5844
5845    if (space->purpose == FIL_TABLESPACE) {
5846      fil_n_pending_tablespace_flushes++;
5847    } else {
5848      fil_n_pending_log_flushes++;
5849      fil_n_log_flushes++;
5850    }
```

this means current node already flushed by someone else

sync buffered write to disk

- `fil/fil0fil.cc:5789, fil_flush()`

```
5857 retry:
5858     if (node->n_pending_flushes > 0) {
5859       /* We want to avoid calling os_file_flush() on
5860       the file twice at the same time, because we do
5861       not know what bugs OS's may contain in file
5862       i/o */
5863
5864       ib_int64_t sig_count =
5865         os_event_reset(node->sync_event);
5866
5867       mutex_exit(&fil_system->mutex);
5868
5869       os_event_wait_low(node->sync_event, sig_count);
5870
5871       mutex_enter(&fil_system->mutex);
5872
5873       if (node->flush_counter >= old_mod_counter) {
5874
5875         goto skip_flush;
5876       }
5877
5878       goto retry;
5879     }
```

earlier sync still running

this means current node already flushed by someone else

sync buffered write to disk

- `fil/fil0fil.cc:5789, fil_flush()`

```
5881     ut_a(node->open);
5882     file = node->handle;
5883     node->n_pending_flushes++;
5884
5885     mutex_exit(&fil_system->mutex);
5886
5887     os_file_flush(file);
5888
5889     mutex_enter(&fil_system->mutex);
5890
5891     os_event_set(node->sync_event);
5892
5893     node->n_pending_flushes--;
5894 skip_flush:
5895     if (node->flush_counter < old_mod_counter) {
5896       node->flush_counter = old_mod_counter;
5897
5898       if (space->is_in_unflushed_spaces
5899           && fil_space_is_flushed(space)) {
5900
5901         space->is_in_unflushed_spaces = false;
5902
5903         UT_LIST_REMOVE(
5904           unflushed_spaces,
5905           fil_system->unflushed_spaces,
5906           space);
5907       }
5908     }
```

call fsync : skip

handle current syncing : change flush_counter, remove from unflushed_spaces

sync buffered write to disk

- `fil/fil0fil.cc:5789, fil_flush()`

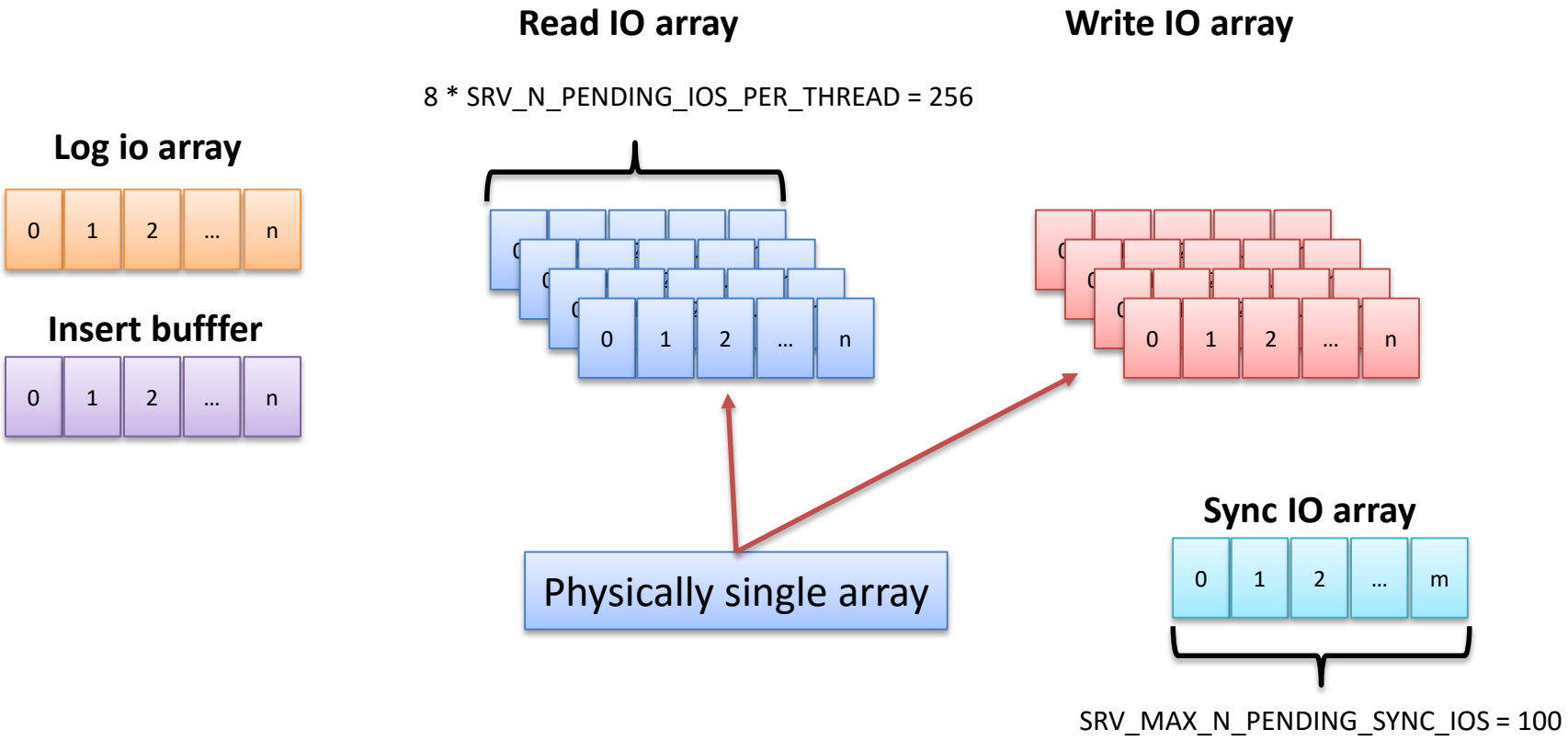
```
5910      if (space->purpose == FIL_TABLESPACE) {
5911        fil_n_pending_tablespace_flushes--;
5912      } else {
5913        fil_n_pending_log_flushes--;
5914      }
5915    }
5916
5917    space->n_pending_flushes--;
5918
5919    mutex_exit(&fil_system->mutex);
5920  }
```

OS related impl. for IO system

- Process IO request
  - fil_io()
    - **os_aio**()
      - sync
      - AIO_NORMAL
      - LOG

InnoDB IO Arrays

- aio thread and array



Do aio – sync case

Read

```
191     *err = fil_io(OS_FILE_READ | wake_later
192             | ignore_nonexistent_pages,
193             sync, space, 0, offset, 0, UNIV_PAGE_SIZE,
194             ((buf_block_t*) bpage)->frame, bpage);
```

dblwr

```
906   fil_io(OS_FILE_WRITE, true, TRX_SYS_SPACE, 0,
907          buf_dblwr->block1, 0, len,
908          (void*) write_buf, NULL);
```



```
5691   /* Queue the aio request */
5692   ret = os_aio(type, mode | wake_later, node->name, node->handle, buf,
5693          offset, len, node, message);
```

* in sync case
type = OS_FILE_READ, OS_FILE_WRITE
mode = OS_AIO_SYNC

Do aio – sync case

- os/os0file.cc:4490, os_aio_func()

```
4489 ibool
4490 os_aio_func(
4491 /*========*/
4492   ulint   type, /*!< in: OS_FILE_READ or OS_FILE_WRITE */
4493   ulint   mode, /*!< in: OS_AIO_NORMAL, ..., possibly ORed
4494         to OS_AIO_SIMULATED_WAKE_LATER: the
4495         last flag advises this function not to wake
4496         i/o-handler threads, but the caller will
4497         do the waking explicitly later, in this
4498         way the caller can post several requests in
4499         a batch; NOTE that the batch must not be
4500         so big that it exhausts the slots in aio
4501         arrays! NOTE that a simulated batch
4502         may introduce hidden chances of deadlocks,
4503         because i/os are not actually handled until
4504         all have been posted: use with great
4505         caution! */
4506   const char* name, /*!< in: name of the file or path as a
4507         null-terminated string */
4508   os_file_t file, /*!< in: handle to a file */
4509   void*   buf,  /*!< in: buffer where to read or from which
4510         to write */
4511   os_offset_t offset, /*!< in: file offset where to read or write */
4512   ulint   n,  /*!< in: number of bytes to read or write */
4513   fil_node_t* message1,/*!< in: message for the aio handler
4514         (can be used to identify a completed
4515         aio operation); ignored if mode is
4516         OS_AIO_SYNC */
4517   void*   message2)/*!< in: message for the aio handler
4518         (can be used to identify a completed
4519         aio operation); ignored if mode is
4520         OS_AIO_SYNC */
4521 {
```

Do aio – sync case

- os/os0file.cc:4490, os_aio_func()

```
4547   if (mode == OS_AIO_SYNC
4548 #ifdef WIN_ASYNC_IO
4549       && !srv_use_native_aio
4550 #endif /* WIN_ASYNC_IO */
4551       ) {
4552     /* This is actually an ordinary synchronous read or write:
4553     no need to use an i/o-handler thread. NOTE that if we use
4554     Windows async i/o, Windows does not allow us to use
4555     ordinary synchronous os_file_read etc. on the same file,
4556     therefore we have built a special mechanism for synchronous
4557     wait in the Windows case.
4558     Also note that the Performance Schema instrumentation has
4559     been performed by current os_aio_func()'s wrapper function
4560     pfs_os_aio_func(). So we would no longer need to call
4561     Performance Schema instrumented os_file_read() and
4562     os_file_write(). Instead, we should use os_file_read_func()
4563     and os_file_write_func() */
4564
4565     if (type == OS_FILE_READ) {
4566       return(os_file_read_func(file, buf, offset, n));
4567     }
4568
4569     ut_ad(!srv_read_only_mode);
4570     ut_a(type == OS_FILE_WRITE);
4571
4572     return(os_file_write_func(name, file, buf, offset, n));
4573   }
```

use ordinary sync io system call
-> actually these functions call pread() and pwrite()

Do aio – async case

batch write

```
781 void
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 {
788   ut_a(bpage);
789   ut_a(buf_page_in_file(bpage));
790
791   const ulint flags = sync
792     ? OS_FILE_WRITE
793     : OS_FILE_WRITE | OS_AIO_SIMULATED_WAKE_LATER;
```

```
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 async case
type = FILE_WRITE with **WAKE_LATER**
mode = sync = **false**

Do aio – async case

- os/os0file.cc:4490, os_aio_func()

```
4489 ibool
4490 os_aio_func(
4491 /*========*/
4492   ulint   type, /*!< in: OS_FILE_READ or OS_FILE_WRITE */
4493   ulint   mode, /*!< in: OS_AIO_NORMAL, ..., possibly ORed
4494         to OS_AIO_SIMULATED_WAKE_LATER: the
4495         last flag advises this function not to wake
4496         i/o-handler threads, but the caller will
4497         do the waking explicitly later, in this
4498         way the caller can post several requests in
4499         a batch; NOTE that the batch must not be
4500         so big that it exhausts the slots in aio
4501         arrays! NOTE that a simulated batch
4502         may introduce hidden chances of deadlocks,
4503         because i/os are not actually handled until
4504         all have been posted: use with great
4505         caution! */
4506   const char* name, /*!< in: name of the file or path as a
4507         null-terminated string */
4508   os_file_t file, /*!< in: handle to a file */
4509   void*   buf,  /*!< in: buffer where to read or from which
4510         to write */
4511   os_offset_t offset, /*!< in: file offset where to read or write */
4512   ulint   n,  /*!< in: number of bytes to read or write */
4513   fil_node_t* message1,/*!< in: message for the aio handler
4514         (can be used to identify a completed
4515         aio operation); ignored if mode is
4516         OS_AIO_SYNC */
4517   void*   message2)/*!< in: message for the aio handler
4518         (can be used to identify a completed
4519         aio operation); ignored if mode is
4520         OS_AIO_SYNC */
4521 {
```

Do aio – async case

- `os/os0file.cc:4490, os_aio_func()`

```
4575 try_again:
4576   switch (mode) {
4577   case OS_AIO_NORMAL:
4578     if (type == OS_FILE_READ) {
4579       array = os_aio_read_array;
4580     } else {
4581       ut_ad(!srv_read_only_mode);
4582       array = os_aio_write_array;
4583     }
4584     break;
4585   case OS_AIO_IBUF:
4586     ut_ad(type == OS_FILE_READ);
4587     /* Reduce probability of deadlock bugs in connection with ibuf:
4588     do not let the ibuf i/o handler sleep */
4589
4590     wake_later = FALSE;
4617   slot = os_aio_array_reserve_slot(type, array, message1, message2, file,
4618           name, buf, offset, n);
```

pick appropriate IO array,
most time it is `os_aio_write_array`.

get a slot for current aio request

Do aio – reserve slot

- `os/os0file.cc:4139, os_aio_array_reserve_slot()`

```
4138 os_aio_slot_t*
4139 os_aio_array_reserve_slot(
4140 /*=======================*/
4141   ulint   type, /*!< in: OS_FILE_READ or OS_FILE_WRITE */
4142   os_aio_array_t* array,  /*!< in: aio array */
4143   fil_node_t* message1,/*!< in: message to be passed along with
4144         the aio operation */
4145   void*   message2,/*!< in: message to be passed along with
4146         the aio operation */
4147   os_file_t file, /*!< in: file handle */
4148   const char* name, /*!< in: name of the file or path as a
4149         null-terminated string */
4150   void*   buf,  /*!< in: buffer where to read or from which
4151         to write */
4152   os_offset_t offset, /*!< in: file offset */
4153   ulint   len)  /*!< in: length of the block to read or write */
4154 {
4155   os_aio_slot_t*  slot = NULL;
4159 #elif defined(LINUX_NATIVE_AIO)
4160
4161   struct iocb*  iocb;
4162   off_t   aio_offset;
4163
```

Do aio – reserve slot

- `os/os0file.cc:4139, os_aio_array_reserve_slot()`

```
4174   /* No need of a mutex. Only reading constant fields */
4175   slots_per_seg = array->n_slots / array->n_segments;
4176
4177   /* We attempt to keep adjacent blocks in the same local
4178   segment. This can help in merging IO requests when we are
4179   doing simulated AIO */
4180   local_seg = (offset >> (UNIV_PAGE_SIZE_SHIFT + 6))
4181     % array->n_segments;
4182
4183 loop:
4184   os_mutex_enter(array->mutex);
4185
4186   if (array->n_reserved == array->n_slots) {
4187     os_mutex_exit(array->mutex);
4188
4189     if (!srv_use_native_aio) {
4190       /* If the handler threads are suspended, wake them
4191       so that we get more slots */
4192
4193       os_aio_simulated_wake_handler_threads();
4194     }
4195
4196     os_event_wait(array->not_full);
4197
4198     goto loop;
4199   }
```

calc. how many segments in current array

get local segment number = this is our request queue no

if IO queue is full

wake io thread

Do aio – reserve slot

- `os/os0file.cc:4139, os_aio_array_reserve_slot()`

```
4201   /* We start our search for an available slot from our preferred
4202   local segment and do a full scan of the array. We are
4203   guaranteed to find a slot in full scan. */
4204   for (i = local_seg * slots_per_seg, counter = 0;
4205        counter < array->n_slots;
4206        i++, counter++) {
4207
4208     i %= array->n_slots;
4209
4210     slot = os_aio_array_get_nth_slot(array, i);
4211
4212     if (slot->reserved == FALSE) {
4213       goto found;
4214     }
4215   }
4216
4217   /* We MUST always be able to get hold of a reserved slot. */
4218   ut_error;
4219
4220 found:
```

Looking for empty slot for io request

Do aio – reserve slot

- `os/os0file.cc:4139, os_aio_array_reserve_slot()`

```
4220 found:
4221   ut_a(slot->reserved == FALSE);
4222   array->n_reserved++;
4223
4224   if (array->n_reserved == 1) {
4225     os_event_reset(array->is_empty);
4226   }
4227
4228   if (array->n_reserved == array->n_slots) {
4229     os_event_reset(array->not_full);
4230   }
4231
4232   slot->reserved = TRUE;
4233   slot->reservation_time = ut_time();
4234   slot->message1 = message1;
4235   slot->message2 = message2;
4236   slot->file     = file;
4237   slot->name     = name;
4238   slot->len      = len;
4239   slot->type     = type;
4240   slot->buf      = static_cast<byte*>(buf);
4241   slot->offset   = offset;
4242   slot->io_already_done = FALSE;
```

set event,
and fill slot values

Do aio – reserve slot

- `os/os0file.cc:4139, os_aio_array_reserve_slot()`

```
4250 #elif defined(LINUX_NATIVE_AIO)
4251
4252   /* If we are not using native AIO skip this part. */
4253   if (!srv_use_native_aio) {
4254     goto skip_native_aio;
4255   }
4256
4257   /* Check if we are dealing with 64 bit arch.
4258   If not then make sure that offset fits in 32 bits. */
4259   aio_offset = (off_t) offset;
4260
4261   ut_a(sizeof(aio_offset) >= sizeof(offset)
4262        || ((os_offset_t) aio_offset) == offset);
4263
4264   iocb = &slot->control;
4265
4266   if (type == OS_FILE_READ) {
4267     io_prep_pread(iocb, file, buf, len, aio_offset);
4268   } else {
4269     ut_a(type == OS_FILE_WRITE);
4270     io_prep_pwrite(iocb, file, buf, len, aio_offset);
4271   }
4272
4273   iocb->data = (void*) slot;
4274   slot->n_bytes = 0;
4275   slot->ret = 0;
4277 skip_native_aio:
4278 #endif /* LINUX_NATIVE_AIO */
4279   os_mutex_exit(array->mutex);
4280
4281   return(slot);
4282 }
```

if use native_aio,
then prepare io request for submit aio

step out

Do aio – async case

- os/os0file.cc:4490, os_aio_func()

```
4619   if (type == OS_FILE_READ) {
4620     if (srv_use_native_aio) {
4621       os_n_file_reads++;
4622       os_bytes_read_since_printout += n;
4623 #ifdef WIN_ASYNC_IO
4624       ret = ReadFile(file, buf, (DWORD) n, &len,
4625                      &(slot->control));
4626
4627 #elif defined(LINUX_NATIVE_AIO)
4628       if (!os_aio_linux_dispatch(array, slot)) {
4629         goto err_exit;
4630       }
4631 #endif /* WIN_ASYNC_IO */
4632     } else {
4633       if (!wake_later) {
4634         os_aio_simulated_wake_handler_thread(
4635           os_aio_get_segment_no_from_slot(
4636             array, slot));
4637       }
4638     }
```

read case

to process native aio request, dispatch current request to kernel

Do aio – async case

- os/os0file.cc:4490, os_aio_func()

```
4639   } else if (type == OS_FILE_WRITE) {
4640     ut_ad(!srv_read_only_mode);
4641     if (srv_use_native_aio) {
4642       os_n_file_writes++;
4643 #ifdef WIN_ASYNC_IO
4644       ret = WriteFile(file, buf, (DWORD) n, &len,
4645           &(slot->control));
4646
4647 #elif defined(LINUX_NATIVE_AIO)
4648       if (!os_aio_linux_dispatch(array, slot)) {
4649         goto err_exit;
4650       }
4651 #endif /* WIN_ASYNC_IO */
4652     } else {
4653       if (!wake_later) {
4654         os_aio_simulated_wake_handler_thread(
4655           os_aio_get_segment_no_from_slot(
4656             array, slot));
4657       }
4658     }
4659   } else {
4660     ut_error;
4661   }
```

write case

to process native aio request, dispatch current request to kernel

Do aio – async case

- os/os0file.cc:4490, os_aio_func()

```
4689   /* aio was queued successfully! */
4690   return(TRUE);
4691
4692 #if defined LINUX_NATIVE_AIO || defined WIN_ASYNC_IO
4693 err_exit:
4694 #endif /* LINUX_NATIVE_AIO || WIN_ASYNC_IO */
4695   os_aio_array_free_slot(array, slot);
4696
4697   if (os_file_handle_error(
4698     name,type == OS_FILE_READ ? "aio read" : "aio write")) {
4699
4700     goto try_again;
4701   }
4702
4703   return(FALSE);
4704 }
```

done!

Do aio – async case

- os/os0file.cc:4895, os_aio_linux_collect()

```
4441 ibool
4442 os_aio_linux_dispatch(
4443 /*====================*/
4444   os_aio_array_t* array,  /*!< in: io request array. */
4445   os_aio_slot_t*  slot) /*!< in: an already reserved slot. */
4446 {
4447   int   ret;
4448   ulint   io_ctx_index;
4449   struct iocb*  iocb;
4450
4451   ut_ad(slot != NULL);
4452   ut_ad(array);
4453
4454   ut_a(slot->reserved);
4455
4456   /* Find out what we are going to work with.
4457   The iocb struct is directly in the slot.
4458   The io_context is one per segment. */
4459
4460   iocb = &slot->control;
4461   io_ctx_index = (slot->pos * array->n_segments) / array->n_slots;
4462
4463   ret = io_submit(array->aio_ctx[io_ctx_index], 1, &iocb);
```

submit io request to kernel

Do aio - aio threads waiting

- Create io-handler-thread
  - During system startup
- AIO waiting
  - after io thread creation, all io threads waiting using `fil_aio_wait()`
  - wake io thread (only in simulated aio thread used, not linux_native_aio)
    - check their io queue(segment)
    - process io requests

Create io threads

- srv/srv0start:1985

```
1974   fsp_init();
1975   log_init();
1976
1977   lock_sys_create(srv_lock_table_size);
1978
1979   /* Create i/o-handler threads: */
1980
1981   for (i = 0; i < srv_n_file_io_threads; ++i) {
1982
1983     n[i] = i;
1984
1985     os_thread_create(io_handler_thread, n + i, thread_ids + i);
1986   }
1987
1988 #ifdef UNIV_LOG_ARCHIVE
1989   if (0 != ut_strcmp(srv_log_group_home_dir, srv_arch_dir)) {
1990     ut_print_timestamp(stderr);
1991     fprintf(stderr, " InnoDB: Error: you must set the log group home dir in my.cnf\n");
1992     ut_print_timestamp(stderr);
1993     fprintf(stderr, " InnoDB: the same as log arch dir.\n");
1994
1995     return(DB_ERROR);
1996   }
1997 #endif /* UNIV_LOG_ARCHIVE */
1998
```

create io threads

Do aio - io-handler creation

- srv/srv0start.cc:466, DECLARE_THREAD(io_handler_thread)

```
465 os_thread_ret_t
466 DECLARE_THREAD(io_handler_thread)(
467 /*===============================*/
468   void* arg)  /*!< in: pointer to the number of the segment in
469       the aio array */
470 {
471   ulint segment;
472
473   segment = *((ulint*) arg);
474
475 #ifdef UNIV_DEBUG_THREAD_CREATION
476   fprintf(stderr, "Io handler thread %lu starts, id %lu\n", segment,
477     os_thread_pf(os_thread_get_curr_id()));
478 #endif
479
480 #ifdef UNIV_PFS_THREAD
481   pfs_register_thread(io_handler_thread_key);
482 #endif /* UNIV_PFS_THREAD */
483
484   while (srv_shutdown_state != SRV_SHUTDOWN_EXIT_THREADS) {
485     fil_aio_wait(segment);
486   }
487
488   /* We count the number of threads in os_thread_exit(). A created
489   thread should always use that to exit and not use return() to exit.
490   The thread actually never comes here because it is exited in an
491   os_event_wait(). */
492
493   os_thread_exit(NULL);
494
495   OS_THREAD_DUMMY_RETURN;
496 }
```

Waiting IO request, until shutdown

Do aio - aio threads waiting

- `fil/fil0fil.cc:5721, fil_aio_wait()`

```
5720 void
5721 fil_aio_wait(
5722 /*==========*/
5723   ulint segment)  /*!< in: the number of the segment in the aio
5724         array to wait for */
5725 {
5726   ibool   ret;
5727   fil_node_t* fil_node;
5728   void*   message;
5729   ulint   type;
5730
5731   ut_ad(fil_validate_skip());
5732
5733   if (srv_use_native_aio) {
5734     srv_set_io_thread_op_info(segment, "native aio handle");
5735 #ifdef WIN_ASYNC_IO
5736     ret = os_aio_windows_handle(
5737       segment, 0, &fil_node, &message, &type);
5738 #elif defined(LINUX_NATIVE_AIO)
5739     ret = os_aio_linux_handle(
5740       segment, &fil_node, &message, &type);
5741 #else
5742     ut_error;
5743     ret = 0; /* Eliminate compiler warning */
5744 #endif /* WIN_ASYNC_IO */
5745   } else {
5746     srv_set_io_thread_op_info(segment, "simulated aio handle");
5747
5748     ret = os_aio_simulated_handle(
5749       segment, &fil_node, &message, &type);
5750   }
```

Do aio - aio threads waiting

- `fil/fil0fil.cc:5721, fil_aio_wait()`

```
5752   ut_a(ret);
5753   if (fil_node == NULL) {
5754     ut_ad(srv_shutdown_state == SRV_SHUTDOWN_EXIT_THREADS);
5755     return;
5756   }
5757
5758   srv_set_io_thread_op_info(segment, "complete io for fil node");
5759
5760   mutex_enter(&fil_system->mutex);
5761
5762   fil_node_complete_io(fil_node, fil_system, type);
5763
5764   mutex_exit(&fil_system->mutex);
5765
5766   ut_ad(fil_validate_skip());
5767
5768   /* Do the i/o handling */
5769   /* IMPORTANT: since i/o handling for reads will read also the insert
5770   buffer in tablespace 0, you have to be very careful not to introduce
5771   deadlocks in the i/o system. We keep tablespace 0 data files always
5772   open, and use a special i/o thread to serve insert buffer requests. */
5773
5774   if (fil_node->space->purpose == FIL_TABLESPACE) {
5775     srv_set_io_thread_op_info(segment, "complete io for buf page");
5776     buf_page_io_complete(static_cast<buf_page_t*>(message));
5777   } else {
5778     srv_set_io_thread_op_info(segment, "complete io for log");
5779     log_io_complete(static_cast<log_group_t*>(message));
5780   }
5781 }
```

done

Do aio – process aio request

- os/os0file.cc:5014, os_aio_linux_handle()

```
5013 ibool
5014 os_aio_linux_handle(
5015 /*=================*/
5016   ulint global_seg, /*!< in: segment number in the aio array
5017         to wait for; segment 0 is the ibuf
5018         i/o thread, segment 1 is log i/o thread,
5019         then follow the non-ibuf read threads,
5020         and the last are the non-ibuf write
5021         threads. */
5022   fil_node_t**message1, /*!< out: the messages passed with the */
5023   void**  message2, /*!< aio request; note that in case the
5024         aio operation failed, these output
5025         parameters are valid and can be used to
5026         restart the operation. */
5027   ulint*  type)   /*!< out: OS_FILE_WRITE or ..._READ */
5028 {
5029   ulint   segment;
5030   os_aio_array_t* array;
5031   os_aio_slot_t*  slot;
5032   ulint   n;
5033   ulint   i;
5034   ibool   ret = FALSE;
5035
5036   /* Should never be doing Sync IO here. */
5037   ut_a(global_seg != ULINT_UNDEFINED);
5038
5039   /* Find the array and the local segment. */
5040   segment = os_aio_get_array_and_local_segment(&array, global_seg);
5041   n = array->n_slots / array->n_segments;
```

Do aio – process aio request

- os/os0file.cc:5014, os_aio_linux_handle()

```
5043   /* Loop until we have found a completed request. */
5044   for (;;) {
5045     ibool any_reserved = FALSE;
5046     os_mutex_enter(array->mutex);
5047     for (i = 0; i < n; ++i) {
5048       slot = os_aio_array_get_nth_slot(
5049         array, i + segment * n);
5050       if (!slot->reserved) {
5051         continue;
5052       } else if (slot->io_already_done) {
5053         /* Something for us to work on. */
5054         goto found;
5055       } else {
5056         any_reserved = TRUE;
5057       }
5058     }
5059
5060     os_mutex_exit(array->mutex);
5061
5062     /* There is no completed request.
5063     If there is no pending request at all,
5064     and the system is being shut down, exit. */
5065     if (UNIV_UNLIKELY
5066         (!any_reserved
5067          && srv_shutdown_state == SRV_SHUTDOWN_EXIT_THREADS)) {
5068       *message1 = NULL;
5069       *message2 = NULL;
5070       return(TRUE);
5071     }
```

if there's nothing to work on, then loops here

Do aio – process aio request

- os/os0file.cc:5014, os_aio_linux_handle()

```
5073     /* Wait for some request. Note that we return
5074     from wait iff we have found a request. */
5075
5076     srv_set_io_thread_op_info(global_seg,
5077       "waiting for completed aio requests");
5078     os_aio_linux_collect(array, segment, n);
5079   }
5080
5081 found:
5082   /* Note that it may be that there are more then one completed
5083   IO requests. We process them one at a time. We may have a case
5084   here to improve the performance slightly by dealing with all
5085   requests in one sweep. */
5086   srv_set_io_thread_op_info(global_seg,
5087         "processing completed aio requests");
5088
5089   /* Ensure that we are scribbling only our segment. */
5090   ut_a(i < n);
5091
5092   ut_ad(slot != NULL);
5093   ut_ad(slot->reserved);
5094   ut_ad(slot->io_already_done);
5095
5096   *message1 = slot->message1;
5097   *message2 = slot->message2;
5098
5099   *type = slot->type;
```

waiting for IO request complete, collect io request status sent by dispatch in os_aio_func()

Do aio – process aio request

- os/os0file.cc:4895, os_aio_linux_collect()

```
4894 void
4895 os_aio_linux_collect(
4896 /*==================*/
4897   os_aio_array_t* array,    /*!< in/out: slot array. */
4898   ulint   segment,  /*!< in: local segment no. */
4899   ulint   seg_size) /*!< in: segment size. */
4900 {
4901   int     i;
4902   int     ret;
4903   ulint     start_pos;
4904   ulint     end_pos;
4905   struct timespec   timeout;
4906   struct io_event*  events;
4907   struct io_context*  io_ctx;
4908
4909   /* sanity checks. */
4910   ut_ad(array != NULL);
4911   ut_ad(seg_size > 0);
4912   ut_ad(segment < array->n_segments);
4913
4914   /* Which part of event array we are going to work on. */
4915   events = &array->aio_events[segment * seg_size];
4916
4917   /* Which io_context we are going to use. */
4918   io_ctx = array->aio_ctx[segment];
4919
4920   /* Starting point of the segment we will be working on. */
4921   start_pos = segment * seg_size;
4922
4923   /* End point. */
4924   end_pos = start_pos + seg_size;
```

Do aio – process aio request

- os/os0file.cc:4895, os_aio_linux_collect()

```
4926 retry:
4927
4928   /* Initialize the events. The timeout value is arbitrary.
4929   We probably need to experiment with it a little. */
4930   memset(events, 0, sizeof(*events) * seg_size);
4931   timeout.tv_sec = 0;
4932   timeout.tv_nsec = OS_AIO_REAP_TIMEOUT;
4933
4934   ret = io_getevents(io_ctx, 1, seg_size, events, &timeout);
4935
4936   if (ret > 0) {
4937     for (i = 0; i < ret; i++) {
4938       os_aio_slot_t*  slot;
4939       struct iocb*  control;
4940
4941       control = (struct iocb*) events[i].obj;
4942       ut_a(control != NULL);
4943
4944       slot = (os_aio_slot_t*) control->data;
4945
4946       /* Some sanity checks. */
4947       ut_a(slot != NULL);
4948       ut_a(slot->reserved);
4949
4950 #if defined(UNIV_AIO_DEBUG)
4951       fprintf(stderr,
4952         "io_getevents[%c]: slot[%p] ctx[%p]"
4953         " seg[%lu]\n",
4954         (slot->type == OS_FILE_WRITE) ? 'w' : 'r',
4955         slot, io_ctx, segment);
4956 #endif
```

IO requests process and get results,

- minimum : 1
- max : current segments (seg_size)
- ret : # of completed events

IO thread that uses linux native aio, blocked here with AIO_REAP_TIMEOUT

Do aio – process aio request

- os/os0file.cc:4895, os_aio_linux_collect()

```
4957
4958        /* We are not scribbling previous segment. */
4959        ut_a(slot->pos >= start_pos);
4960
4961        /* We have not overstepped to next segment. */
4962        ut_a(slot->pos < end_pos);
4963
4964        /* Mark this request as completed. The error handling
4965        will be done in the calling function. */
4966        os_mutex_enter(array->mutex);
4967        slot->n_bytes = events[i].res;
4968        slot->ret = events[i].res2;
4969        slot->io_already_done = TRUE;
4970        os_mutex_exit(array->mutex);
4971      }
4972      return;
4973    }
```

Process - done

Do aio – process aio request

- os/os0file.cc:4895, os_aio_linux_collect()

```
4975   if (UNIV_UNLIKELY(srv_shutdown_state == SRV_SHUTDOWN_EXIT_THREADS)) {
4976     return;
4977   }
4978
4979   /* This error handling is for any error in collecting the
4980   IO requests. The errors, if any, for any particular IO
4981   request are simply passed on to the calling routine. */
4982
4983   switch (ret) {
4984   case -EAGAIN:
4985     /* Not enough resources! Try again. */
4986   case -EINTR:
4987     /* Interrupted! I have tested the behaviour in case of an
4988     interrupt. If we have some completed IOs available then
4989     the return code will be the number of IOs. We get EINTR only
4990     if there are no completed IOs and we have been interrupted. */
4991   case 0:
4992     /* No pending request! Go back and check again. */
4993     goto retry;
4994   }
4995
4996   /* All other errors should cause a trap for now. */
4997   ut_print_timestamp(stderr);
4998   fprintf(stderr,
4999     " InnoDB: unexpected ret_code[%d] from io_getevents()!\n",
5000     ret);
5001   ut_error;
5002 }
```

nothing have been processed - retry!

Do aio – process aio request

- os/os0file.cc:5014, os_aio_linux_handle()

```
5073     /* Wait for some request. Note that we return
5074     from wait iff we have found a request. */
5075
5076     srv_set_io_thread_op_info(global_seg,
5077       "waiting for completed aio requests");
5078     os_aio_linux_collect(array, segment, n);
5079   }
5080
5081 found:
5082   /* Note that it may be that there are more then one completed
5083   IO requests. We process them one at a time. We may have a case
5084   here to improve the performance slightly by dealing with all
5085   requests in one sweep. */
5086   srv_set_io_thread_op_info(global_seg,
5087         "processing completed aio requests");
5088
5089   /* Ensure that we are scribbling only our segment. */
5090   ut_a(i < n);
5091
5092   ut_ad(slot != NULL);
5093   ut_ad(slot->reserved);
5094   ut_ad(slot->io_already_done);
5095
5096   *message1 = slot->message1;
5097   *message2 = slot->message2;
5098
5099   *type = slot->type;
```

waiting for IO request complete, collect io request status sent by dispatch in os_aio_func()

Do aio – process aio request

- `os/os0file.cc:5014, os_aio_linux_handle()`

```
5101   if (slot->ret == 0 && slot->n_bytes == (long) slot->len) {
5102
5103     ret = TRUE;
5104   } else {
5105     errno = -slot->ret;
5106
5107     /* os_file_handle_error does tell us if we should retry
5108     this IO. As it stands now, we don't do this retry when
5109     reaping requests from a different context than
5110     the dispatcher. This non-retry logic is the same for
5111     windows and linux native AIO.
5112     We should probably look into this to transparently
5113     re-submit the IO. */
5114     os_file_handle_error(slot->name, "Linux aio");
5115
5116     ret = FALSE;
5117   }
5118
5119   os_mutex_exit(array->mutex);
5120
5121   os_aio_array_free_slot(array, slot);
5122
5123   return(ret);
5124 }
```

io request processed correctly

error handling

step out

run sysbench

EXTRA

Sysbench

- Sysbench
  - one of famous benchmark tool
    - File I/O performance
    - Scheduler performance
    - Memory allocation and transfer speed
    - POSIX thread impl. performance
    - Database Server performance (OLTP)
      - MySQL
      - PostgreSQL
      - Drizzle
      - …

Sysbench

- Download from Launchpad

```
$> bzr branch lp:sysbench
```

- Build

```
$> cd sysbench
$>./autogen.sh
$>./configure \
  --without-drizzle \
  --with-mysql-includes=/usr/local/mysql/include/ \
  --with-mysql-libs=/usr/local/mysql/lib/
$> make
```

use appropriate path
(where you installed)

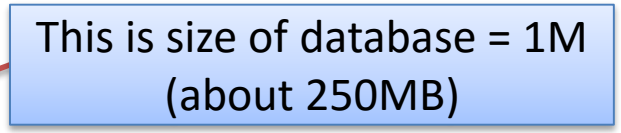
Sysbench

- Prepare database

```
$> mysqladmin -uroot create sbtest
```

- Load database (population database files)

```
 5 SYSBENCH_TESTS="delete.lua \
 6                 insert.lua \
 7                 oltp_complex_ro.lua \
 8                 oltp_complex_rw.lua \
 9                 oltp_simple.lua \
10                 select.lua \
11                 update_index.lua \
12                 update_non_index.lua"
13
14 NUM_THREADS=16
15
16 SYSBENCH_DIR=${HOME}/sysbench/
17 TEST_DIR=${SYSBENCH_DIR}/sysbench/tests/db
18
19 ./sysbench \
20   --test=${TEST_DIR}/oltp.lua \
21   --oltp-table-size=10000000 \
22   --max-time=600 \
23   --max-requests=0 \
24   --mysql-table-engine=InnoDB \
25   --mysql-user=root \
26   --mysql-engine-trx=yes \
27   --num-threads=$NUM_THREADS \
28   prepare
```



Sysbench

- Running benchmark

```
14 NUM_THREADS=16
15 TEST_DIR=${HOME}/sysbench/sysbench/tests/db
16
17 ./sysbench \
18   --test=${TEST_DIR}/oltp.lua \
19   --oltp-table-size=10000000 \
20   --max-time=600 \
21   --max-requests=0 \
22   --mysql-table-engine=InnoDB \
23   --mysql-user=root \
24   --mysql-engine-trx=yes \
25   --num-threads=$NUM_THREADS \
26   run
```

Sysbench

- Results

```
Threads started!

OLTP test statistics:
    queries performed:
        read:                            5390714
        write:                           1540204
        other:                           770102
        total:                           7701020
    transactions:                        385051 (3208.66 per sec.)
    read/write requests:                 6930918 (57755.89 per sec.)
    other operations:                    770102 (6417.32 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          120.0037s
    total number of events:              385051
    total time taken by event execution: 1919.5407s
    response time:
         min:                                  1.52ms
         avg:                                  4.99ms
         max:                                111.25ms
         approx.  95 percentile:               8.94ms

Threads fairness:
    events (avg/stddev):           24065.6875/120.68
    execution time (avg/stddev):   119.9713/0.00
```

Q&A

- Any Questions ?

Reference

- Source Code : MySQL Community Server 5.6.21