InnoDB Flushing and Checkpoints

Mijin An

meeeeejin@gmail.com

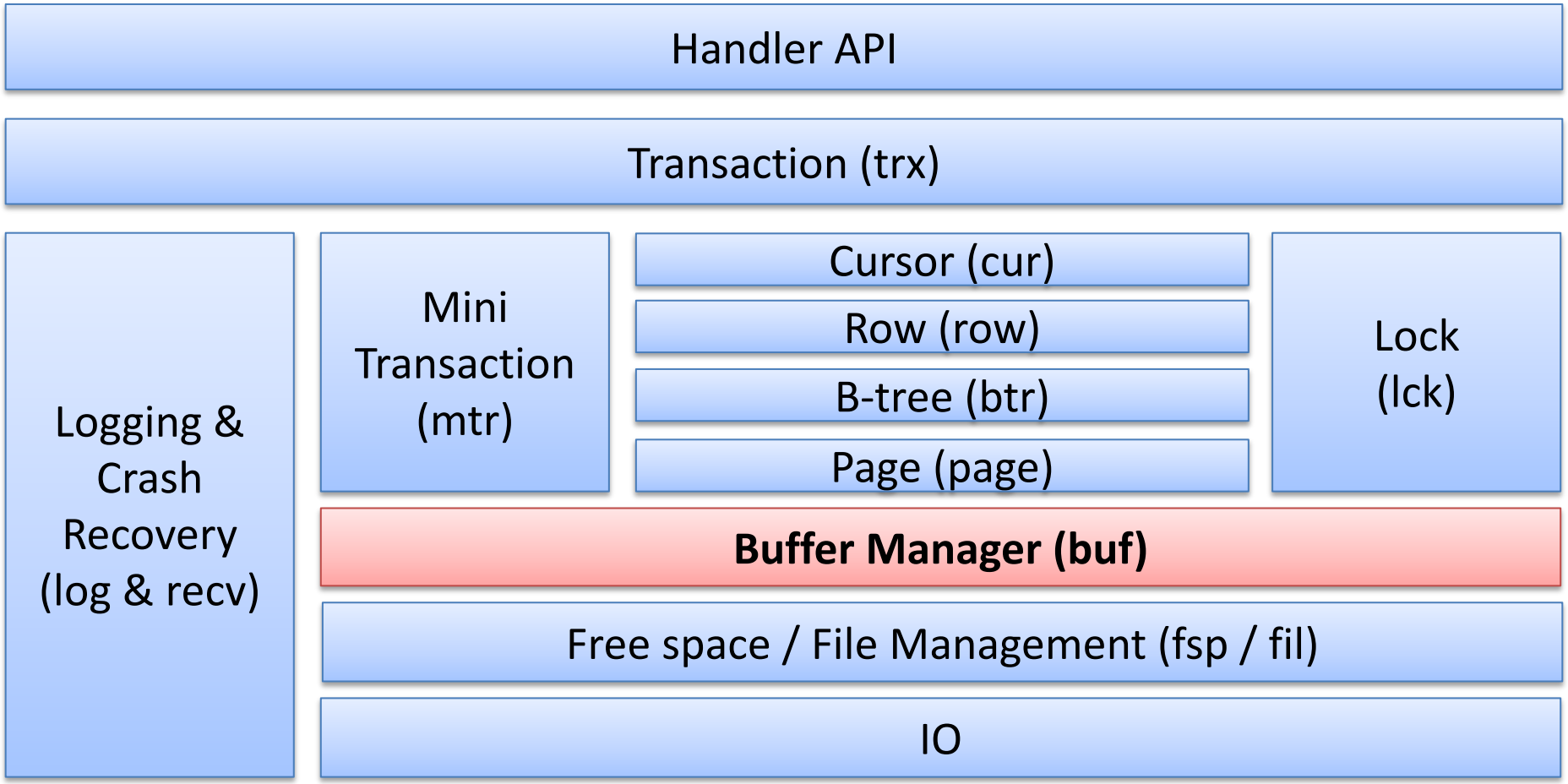
Contents

- Overview
- Page Cleaner Thread
- LRU List Flushing
- Flush List Flushing
- MySQL Checkpoints

OVERVIEW

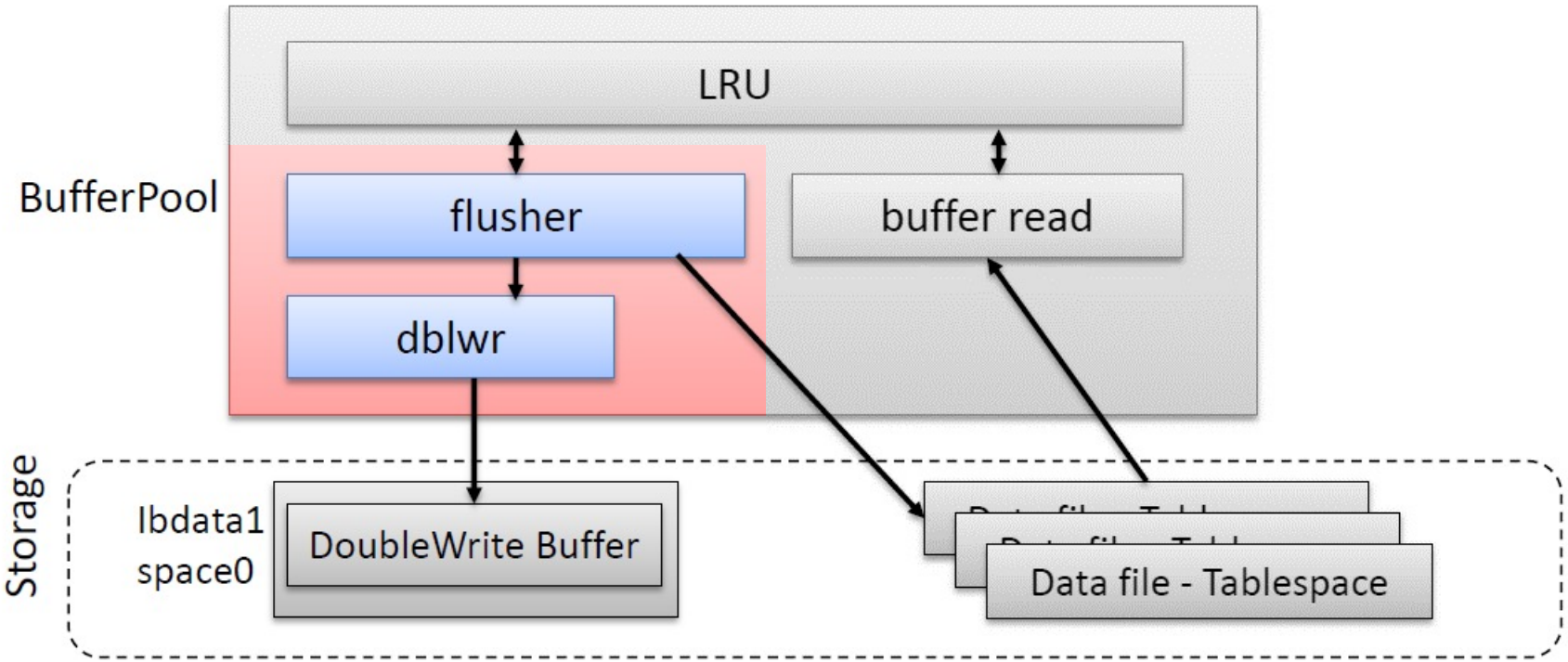
Overview

- InnoDB Architecture

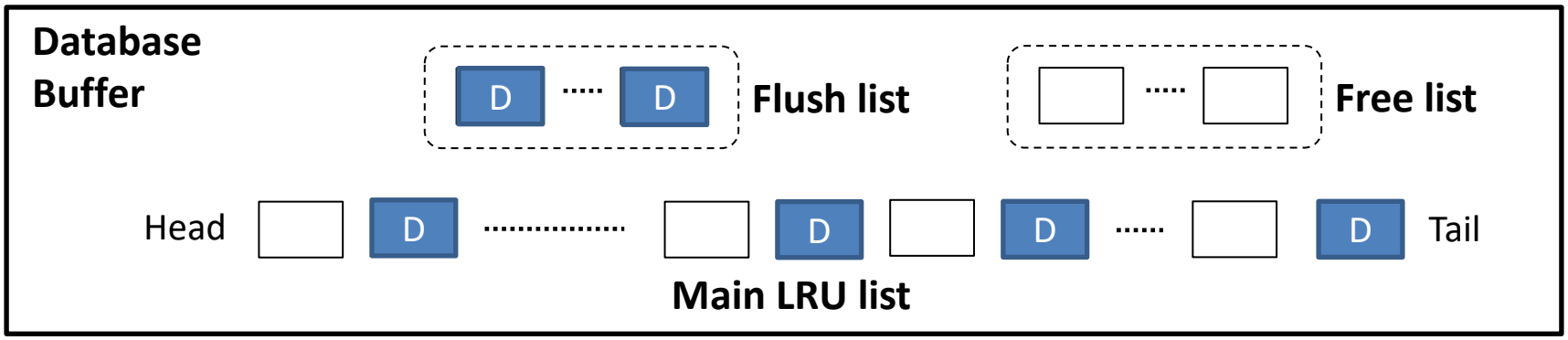


Overview

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



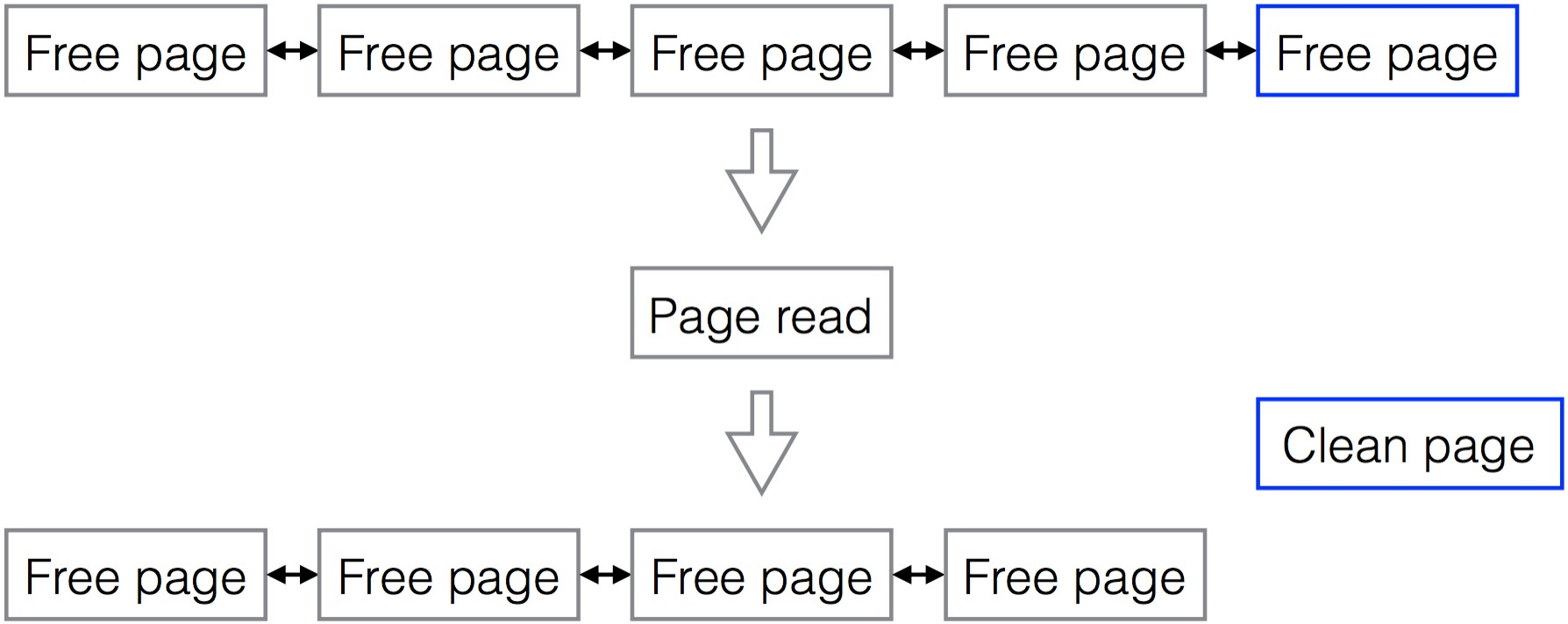
Lists of Buffer Blocks



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

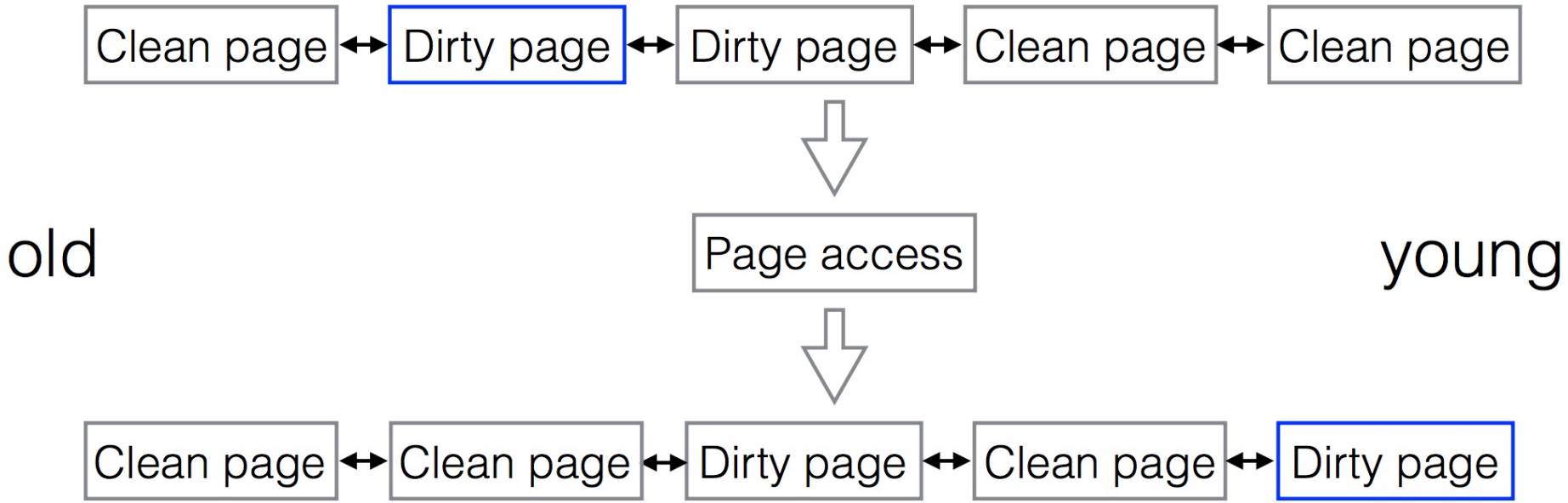
Free list

- **Free list** for having free space in the buffer pool to read currently non-present pages. Reading:



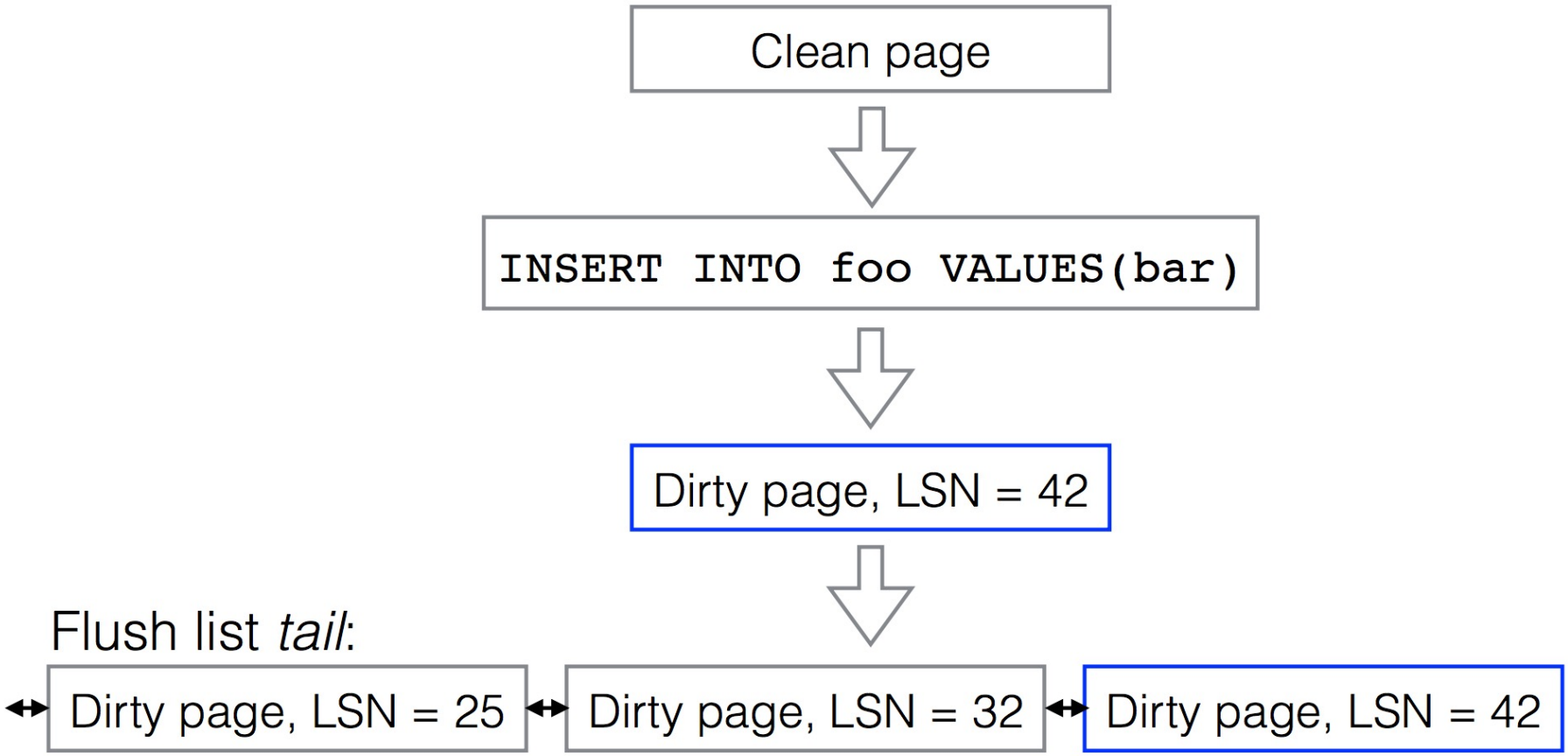
LRU list

- **LRU list** for deciding which pages to evict
  - Preventing eviction for recently-used pages (making them young)



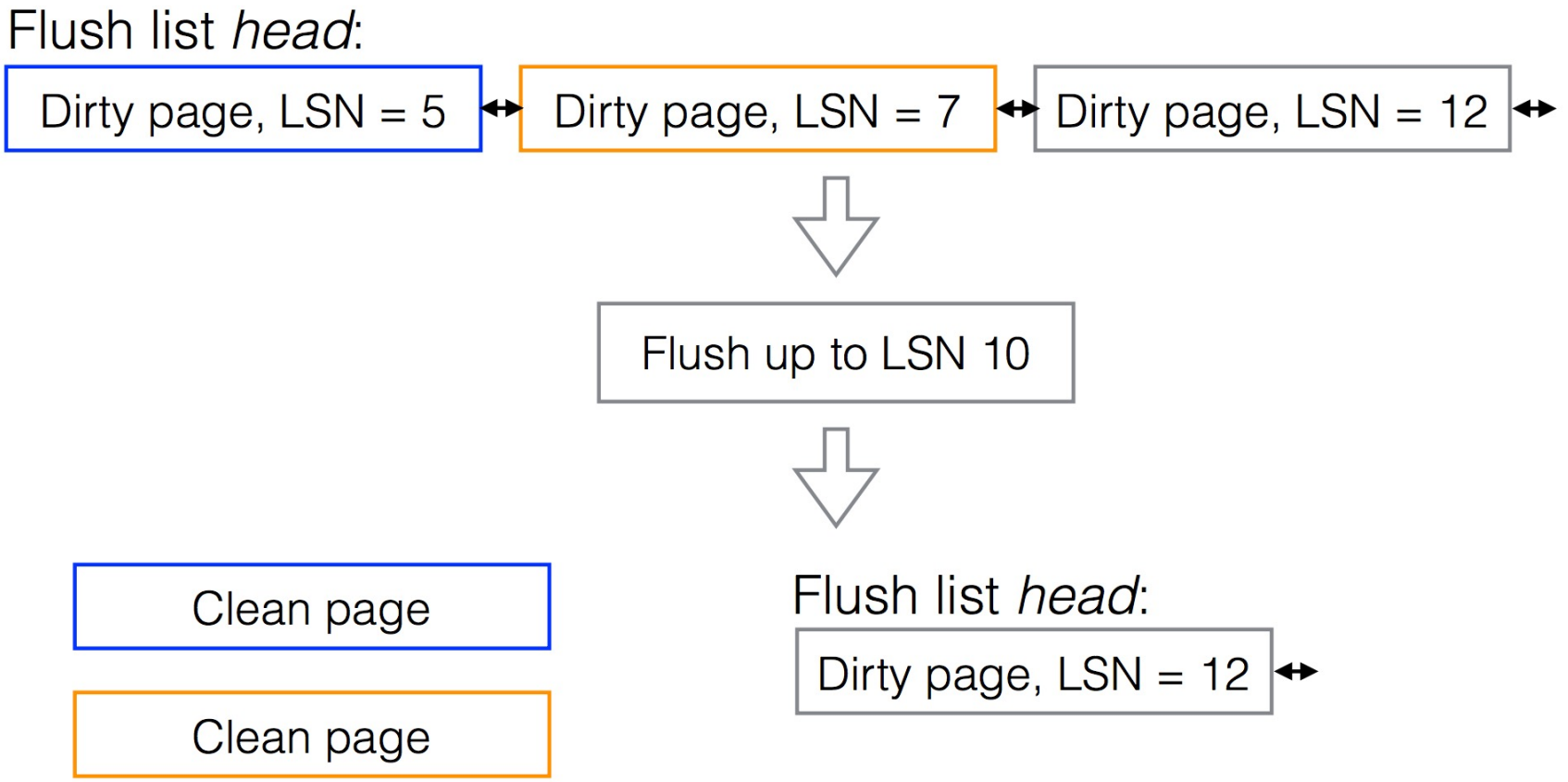
Flush list

- **Flush list** for dirty page management. Dirtying:



Flush list

- **Flush list** for dirty page management. Flushing:



Flushing

- To write dirty pages to disk in the background, that had been buffered in a memory area

- InnoDB has **limited** space in the buffer pool and redo log
- InnoDB tries to avoid the need for synchronous I/O by flushing dirty pages continually, keeping a reserve of clean or free spaces that can be replaced without having to be flushed

Flushing Challenge

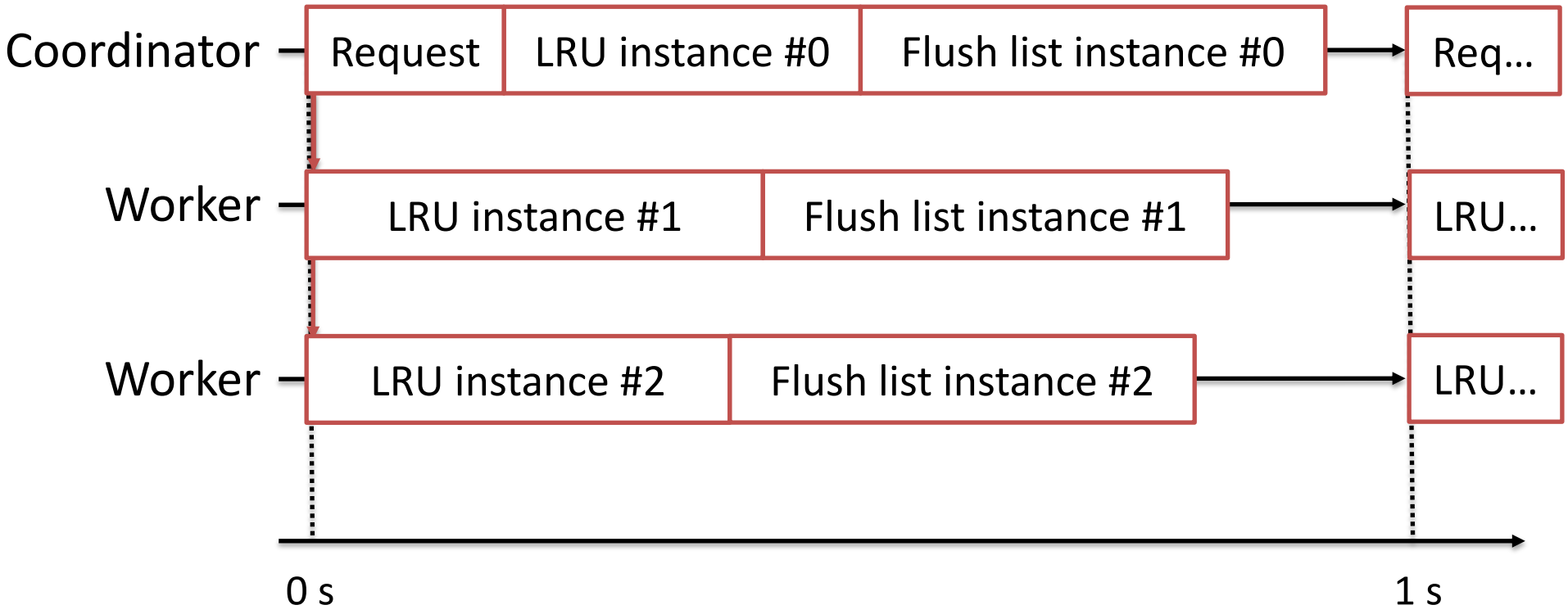
- Users like uniform performance
- Flush enough not to run out of **log** space
- Do not flush too aggressively to impact **performance**
- That is, it is a hard problem to balance
  - Flush too much slow
    - Compete with I/O which must happen
    - Loose possibility of optimization
  - Delay too much
    - Might have to do too much I/O in the future
    - Potentially cause “stalls” or performance dips

PAGE CLEANER THREAD

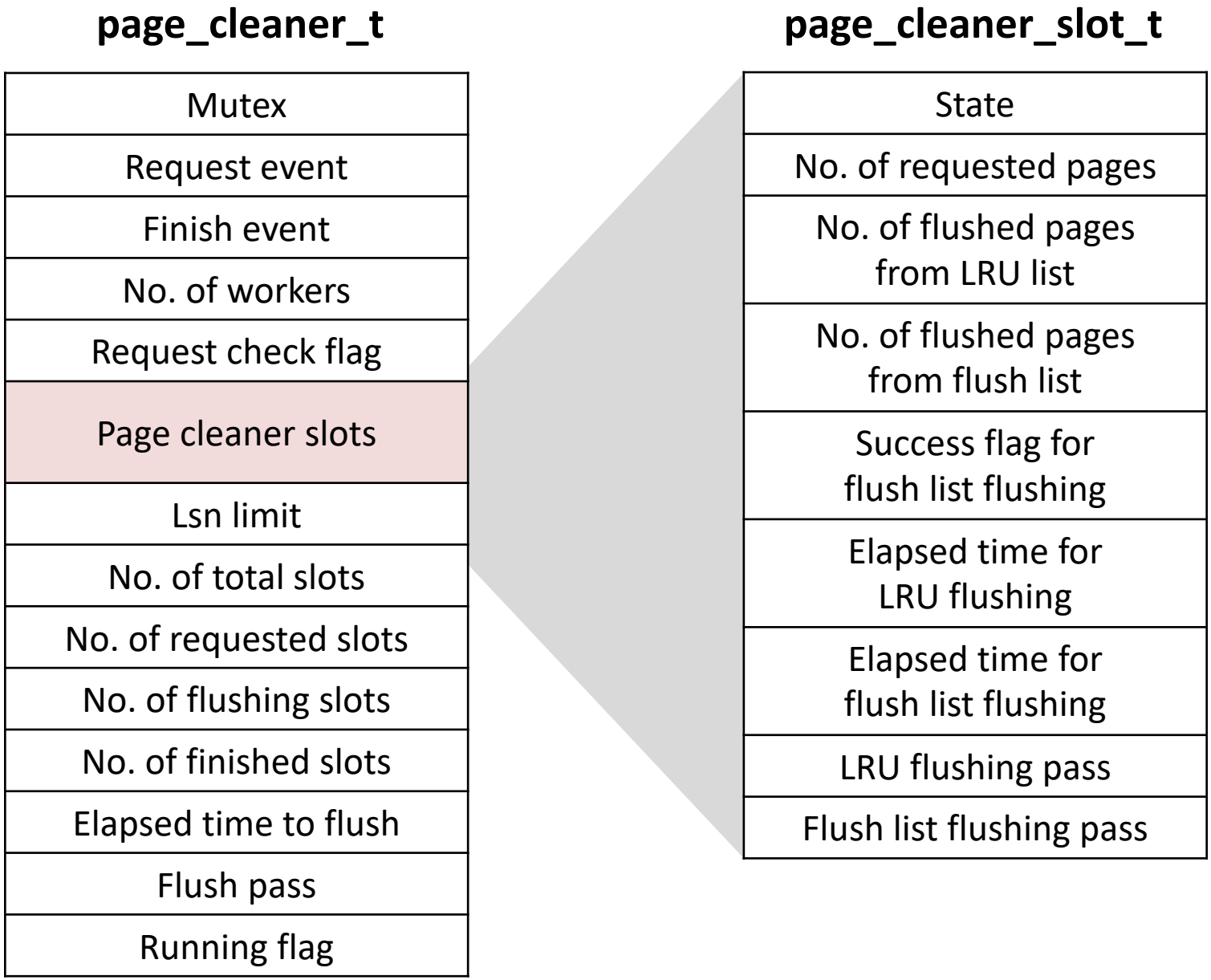
Page Cleaner Thread(s)

- Handles all types of background flushing
  - Flushes pages from end of LRU list
  - Flushes pages from flush list
- Wakes up once per second
- In some cases flushing can be done from user threads
- Multiple threads available in MySQL 5.7+

Multi-threaded flushing



Page Cleaner Thread



Page Cleaner Struct

- buf/buf0flu.cc: page_cleaner_t

```
162  /** Page cleaner structure common for all threads */
163  struct page_cleaner_t {
164    ib_mutex_t mutex;          /*!< mutex to protect whole of
165                               page_cleaner_t struct and
166                               page_cleaner_slot_t slots. */
167    os_event_t is_requested;   /*!< event to activate worker
168                               threads. */
169    os_event_t is_finished;    /*!< event to signal that all
170                               slots were finished. */
171    volatile ulint n_workers; /*!< number of worker threads
172                               in existence */
173    bool requested;            /*!< true if requested pages
174                               to flush */
175    lsn_t lsn_limit;           /*!< upper limit of LSN to be
176                               flushed */
177    ulint n_slots;             /*!< total number of slots */
```

Page Cleaner Struct

- buf/buf0flu.cc: page_cleaner_t

```
178     ulint n_slots_requested;
179     /*!< number of slots
180     in the state
181     PAGE_CLEANER_STATE_REQUESTED */
182     ulint n_slots_flushing;
183     /*!< number of slots
184     in the state
185     PAGE_CLEANER_STATE_FLUSHING */
186     ulint n_slots_finished;
187     /*!< number of slots
188     in the state
189     PAGE_CLEANER_STATE_FINISHED */
190     ulint flush_time;           /*!< elapsed time to flush
191                                 requests for all slots */
192     ulint flush_pass;           /*!< count to finish to flush
193                                 requests for all slots */
194     page_cleaner_slot_t *slots; /*!< pointer to the slots */
195     bool is_running;            /*!< false if attempt
196                                 to shutdown */
```

Page Cleaner Struct

- buf/buf0flu.cc: page_cleaner_slot_t

```
126  /** Page cleaner request state for each buffer pool instance */
127  struct page_cleaner_slot_t {
128    page_cleaner_state_t state; /*!< state of the request.
129                                protected by page_cleaner_t::mutex
130                                if the worker thread got the slot and
131                                set to PAGE_CLEANER_STATE_FLUSHING,
132                                n_flushed_lru and n_flushed_list can be
133                                updated only by the worker thread */
134    /* This value is set during state==PAGE_CLEANER_STATE_NONE */
135    ulint n_pages_requested;
136    /*!< number of requested pages
137    for the slot */
138    /* These values are updated during state==PAGE_CLEANER_STATE_FLUSHING,
139    and commited with state==PAGE_CLEANER_STATE_FINISHED.
140    The consistency is protected by the 'state' */
141    ulint n_flushed_lru;
142    /*!< number of flushed pages
143    by LRU scan flushing */
```

Page Cleaner Struct

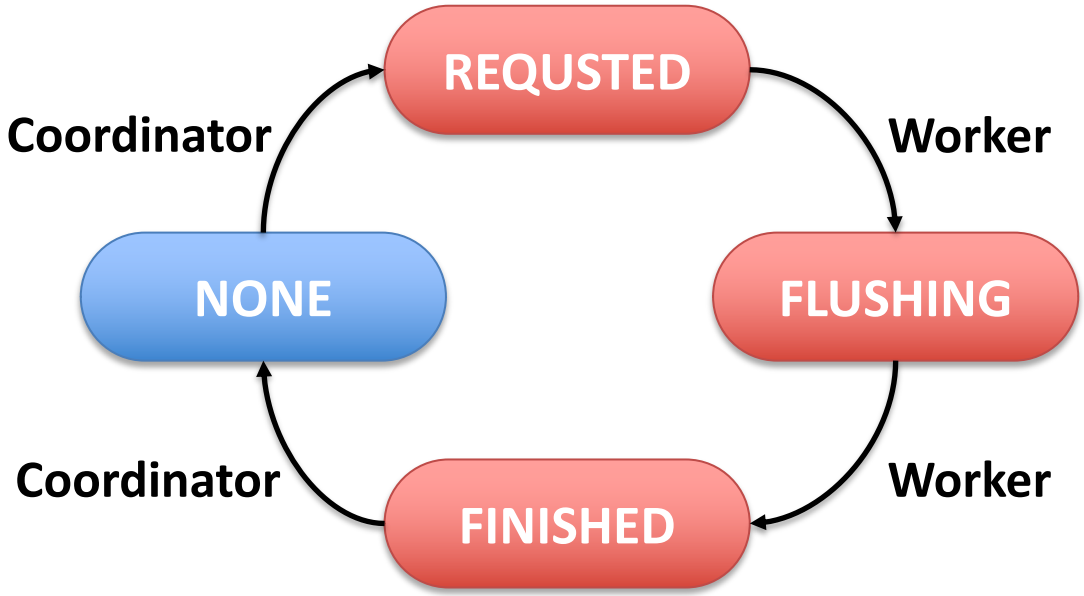
- buf/buf0flu.cc: page_cleaner_slot_t

```
144     ulint n_flushed_list;
145     /*!< number of flushed pages
146     by flush_list flushing */
147     bool succeeded_list;
148     /*!< true if flush_list flushing
149     succeeded. */
150     ulint flush_lru_time;
151     /*!< elapsed time for LRU flushing */
152     ulint flush_list_time;
153     /*!< elapsed time for flush_list
154     flushing */
155     ulint flush_lru_pass;
156     /*!< count to attempt LRU flushing */
157     ulint flush_list_pass;
158     /*!< count to attempt flush_list
159     flushing */
160   };
```

Page Cleaner Struct

- buf/buf0flu.cc: page_cleaner_state_t

```
110  /** State for page cleaner array slot */
111  enum page_cleaner_state_t {
112    /** Not requested any yet.
113    Moved from FINISHED by the coordinator. */
114    PAGE_CLEANER_STATE_NONE = 0,
115    /** Requested but not started flushing.
116    Moved from NONE by the coordinator. */
117    PAGE_CLEANER_STATE_REQUESTED,
118    /** Flushing is on going.
119    Moved from REQUESTED by the worker. */
120    PAGE_CLEANER_STATE_FLUSHING,
121    /** Flushing was finished.
122    Moved from FLUSHING by the worker. */
123    PAGE_CLEANER_STATE_FINISHED
124  };
```



Page Cleaner Init

- buf/buf0flu.cc: buf_flush_page_cleaner_init()

```
192  static page_cleaner_t*  page_cleaner = NULL;
                    ...
2743 /** Initialize page_cleaner.
2744 @param[in]     n_page_cleaners Number of page cleaner threads to create */
2745 void
2746 buf_flush_page_cleaner_init(size_t n_page_cleaners)
2747 {
2748         ut_ad(page_cleaner == NULL);
2749
2750         page_cleaner = static_cast<page_cleaner_t*>(
2751                 ut_zalloc_nokey(sizeof(*page_cleaner)));
2752
2753         mutex_create(LATCH_ID_PAGE_CLEANER, &page_cleaner->mutex);
2754
2755         page_cleaner->is_requested = os_event_cre
2756         page_cleaner->is_finished = os_event_crea
2757
2758         page_cleaner->n_slots = static_cast<ulint>(srv_buf_pool_instances);
```

Global variable; Coordinator

Create slots as many as the number of BP instances

Page Cleaner Init

- buf/buf0flu.cc: buf_flush_page_cleaner_init()

```
2760        page_cleaner->slots = static_cast<page_cleaner_slot_t*>(
2761                ut_zalloc_nokey(page_cleaner->n_slots
2762                                * sizeof(*page_cleaner->slots)));
2763
2764        ut_d(page_cleaner->n_disabled_debug = 0);
2765
2766        page_cleaner->is_running = true;
2767
2768        os_thread_create(
2769                page_flush_coordinator_thread_key,
2770                buf_flush_page_coordinator_thread,
2771                n_page_cleaners);
2772
2773        /* Make sure page cleaner is active. */
2774
2775        while (!buf_page_cleaner_is_active) {
2776                os_thread_sleep(10000);
2777        }
2778  }
```

Create a page cleaner thread (coordinator)

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: buf_flush_page_coordinator_thread()

```
2882  /** Thread tasked with flushing dirty pages from the buffer pools.
2883  As of now we'll have only one coordinator.
2884  @param[in]      n_page_cleaners Number of page cleaner threads to create */
2885  static void buf_flush_page_coordinator_thread(size_t n_page_cleaners) {
2886    ulint next_loop_time = ut_time_ms() + 1000;
2887    ulint n_flushed = 0;
2888    ulint last_activity = srv_get_activity_count();
2889    ulint last_pages = 0;
2890
2891    my_thread_init();
```

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: buf_flush_page_coordinator_thread()

```
2906    buf_page_cleaner_is_active = true;
2907
2908    /* We start from 1 because the coordinator thread is part of the
2909    same set */
2910
2911    for (size_t i = 1; i < n_page_cleaners; ++i) {
2912      os_thread_create(page_flush_thread_key, buf_flush_page_cleaner_thread);
2913    }
```

Create page cleaner threads (workers)

Page Cleaner: Worker Thread

- buf/buf0flu.cc: buf_flush_page_cleaner_thread()

```
3228  /** Worker thread of page_cleaner. */
3229  static void buf_flush_page_cleaner_thread() {
3230    my_thread_init();
3231    mutex_enter(&page_cleaner->mutex);
3232    ++page_cleaner->n_workers;
3233    mutex_exit(&page_cleaner->mutex);
3234
3235  #ifdef UNIV_LINUX
3236    /* linux might be able to set different setting for each thread
3237    worth to try to set high priority for page cleaner threads */
3238    if (buf_flush_page_cleaner_set_priority(buf_flush_page_cleaner_priority)) {
3239      ib::info(ER_IB_MSG_129)
3240          << "page_cleaner worker priority: " << buf_flush_page_cleaner_priority;
3241    }
3242  #endif /* UNIV_LINUX */
```

Page Cleaner: Worker Thread

- buf/buf0flu.cc: buf_flush_page_cleaner_thread()

```
3244    for (;;) {
3245      os_event_wait(page_cleaner->is_requested);
3246
3247      ut_d(buf_flush_page_cleaner_disabled_loop());
3248
3249      if (!page_cleaner->is_running) {
3250        break;
3251      }
3252
3253      pc_flush_slot();
3254    }
3255
3256    mutex_enter(&page_cleaner->mutex);
3257    --page_cleaner->n_workers;
3258    mutex_exit(&page_cleaner->mutex);
3259    my_thread_end();
3260  }
```

Wait for the **REQUEST** event

Run until shutdown

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: buf_flush_page_coordinator_thread()

```
2953    os_event_wait(buf_flush_event);
2954
2955    ulint ret_sleep = 0;
2956    ulint n_evicted = 0;
2957    ulint n_flushed_last = 0;
2958    ulint warn_interval = 1;
2959    ulint warn_count = 0;
2960    int64_t sig_count = os_event_reset(buf_flush_event);
2961
2962    while (srv_shutdown_state == SRV_SHUTDOWN_NONE) {
2963      /* The page_cleaner skips sleep if the server is
2964      idle and there are no pending IOs in the buffer pool
2965      and there is work to do. */
2966      if (srv_check_activity(last_activity) || buf_get_n_pending_read_ios() ||
2967          n_flushed == 0) {
2968        ret_sleep = pc_sleep_if_needed(next_loop_time, sig_count);
```

Run until shutdown

If the previous flushing time has exceeded the specified time(1s), set OS_SYNC_TIME_EXCEEDED

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: buf_flush_page_coordinator_thread()

```
2979    sig_count = os_event_reset(buf_flush_event);
                                  ...
3014    if (ret_sleep != OS_SYNC_TIME_EXCEEDED && srv_flush_sync &&
3015        buf_flush_sync_lsn > 0) {
3016      /* woke up for flush_sync */
3017      mutex_enter(&page_cleaner->mutex);
3018      lsn_t lsn_limit = buf_flush_sync_lsn;
3019      buf_flush_sync_lsn = 0;
3020      mutex_exit(&page_cleaner->mutex);
3021
3022      /* Request flushing for threads */
3023      pc_request(ULINT_MAX, lsn_limit);
3024
3025      ulint tm = ut_time_ms();
3026
3027      /* Coordinator also treats requests */
3028      while (pc_flush_slot() > 0) {
3029      }
```

1st case: **Emergency**! We have to do a *synchronous* flush, because the **oldest dirty page is too old.**

Wake Page Cleaner for I/O Burst

- log/log0chkp.cc: log_preflush_pool_modified_pages()

```
666  static void log_preflush_pool_modified_pages(const log_t &log,
667                                               lsn_t new_oldest) {
668    /* A flush is urgent: we have to do a synchronous flush,
669    because the oldest dirty page is too old.
670
671    Note, that this could fire even if we did not run out
672    of space in log files (users still may write to redo). */
673
674    if (new_oldest == LSN_MAX
675        /* Forced flush request is processed by page_cleaner, if
676        it's not active, then we must do flush ourselves. */
677        || !buf_page_cleaner_is_active
678        /* Reason unknown. */
679        || srv_is_being_started) {
680      buf_flush_sync_all_buf_pools();
681
682    } else {
683      new_oldest += log_buffer_flush_order_lag(log);
```

Wake Page Cleaner for I/O Burst

- log/log0chkp.cc: log_preflush_pool_modified_pages()

```
685       /* better to wait for being flushed by page cleaner */
686       if (srv_flush_sync) {
687         /* wake page cleaner for IO burst */
688         buf_flush_request_force(new_oldest);
689       }
690
691       buf_flush_wait_flushed(new_oldest);
692     }
693   }
```

Wake Page Cleaner for I/O Burst

- buf/buf0flu.cc: buf_flush_request_force()

```
3285  /** Request IO burst and wake page_cleaner up.
3286  @param[in]      lsn_limit       upper limit of LSN to be flushed */
3287  void buf_flush_request_force(lsn_t lsn_limit) {
3288    ut_a(buf_page_cleaner_is_active);
3289
3290    /* adjust based on lsn_avg_rate not to get old */
3291    lsn_t lsn_target = lsn_limit + lsn_avg_rate * 3;
3292
3293    mutex_enter(&page_cleaner->mutex);
3294    if (lsn_target > buf_flush_sync_lsn) {
3295      buf_flush_sync_lsn = lsn_target;
3296    }
3297    mutex_exit(&page_cleaner->mutex);
3298
3299    os_event_set(buf_flush_event);
3300  }
```

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: buf_flush_page_coordinator_thread()

```
3051        } else if (srv_check_activity(last_activity)) {
3052          ulint n_to_flush;
3053          lsn_t lsn_limit = 0;
3054
3055          /* Estimate pages from flush_list to be flushed */
3056          if (ret_sleep == OS_SYNC_TIME_EXCEEDED) {
3057            last_activity = srv_get_activity_count();
3058            n_to_flush =
3059                page_cleaner_flush_pages_recommendation(&lsn_limit, last_pages);
3060          } else {
3061            n_to_flush = 0;
3062          }
3063
3064          /* Request flushing for threads */
3065          pc_request(n_to_flush, lsn_limit);
3066
3067          ulint tm = ut_time_ms();
```

2nd case: Something has been changed on server! (normal)

If *OS_SYNC_TIME_EXCEEDED* is set, decides the number of pages recommended to flush from the **flush list**

Expected # of flushed pages from flush list

- buf/buf0flu.cc: page_cleaner_flush_pages_recommendation()

```
2276  /** This function is called approximately once every second by the
2277   page_cleaner thread. Based on various factors it decides if there is a
2278   need to do flushing.
2279   @return number of pages recommended to be flushed
2280   @param lsn_limit       pointer to return LSN up to which flushing must happen
2281   @param last_pages_in   the number of pages flushed by the last flush_list
2282                           flushing. */
2283  static ulint page_cleaner_flush_pages_recommendation(lsn_t *lsn_limit,
2284                                                       ulint last_pages_in) {
2285    static lsn_t prev_lsn = 0;
2286    static ulint sum_pages = 0;
2287    static ulint avg_page_rate = 0;
2288    static ulint n_iterations = 0;
2289    static time_t prev_time;
2290    lsn_t oldest_lsn;
2291    lsn_t cur_lsn;
2292    lsn_t age;
2293    lsn_t lsn_rate;
```

Expected # of flushed pages from flush list

- buf/buf0flu.cc: page_cleaner_flush_pages_recommendation()

```
2299    cur_lsn = log_buffer_dirty_pages_added_up_to_lsn(*log_sys);
2300
2301    if (prev_lsn == 0) {
2302      /* First time around. */
2303      prev_lsn = cur_lsn;
2304      prev_time = ut_time();
2305      return (0);
2306    }
2307
2308    if (prev_lsn == cur_lsn) {
2309      return (0);
2310    }
2311
2312    sum_pages += last_pages_in;
2313
2314    time_t curr_time = ut_time();
2315    double time_elapsed = difftime(curr_time, prev_time);
```

Get current LSN

Expected # of flushed pages from flush list

- buf/buf0flu.cc: page_cleaner_flush_pages_recommendation()

```
2317    /* We update our variables every srv_flushing_avg_loops
2318    iterations to smooth out transition in workload. */
2319    if (++n_iterations >= srv_flushing_avg_loops ||
2320        time_elapsed >= srv_flushing_avg_loops) {
2321      if (time_elapsed < 1) {
2322        time_elapsed = 1;
2323      }
2324
2325      avg_page_rate = static_cast<ulint>(
2326          ((static_cast<double>(sum_pages) / time_elapsed) + avg_page_rate) / 2);
2327
2328      /* How much LSN we have generated since last call. */
2329      lsn_rate = static_cast<lsn_t>(static_cast<double>(cur_lsn - prev_lsn) /
2330                                    time_elapsed);
2331
2332      lsn_avg_rate = (lsn_avg_rate + lsn_rate) / 2;
```

Calculate average # of pages flushed per second

Calculate average generation rate of LSN per second

Expected # of flushed pages from flush list

- buf/buf0flu.cc: page_cleaner_flush_pages_recommendation()

```
2334    /* aggregate stats of all slots */
2335    mutex_enter(&page_cleaner->mutex);
2336
2337    ulint flush_tm = page_cleaner->flush_time;
2338    ulint flush_pass = page_cleaner->flush_pass;
2339
2340    page_cleaner->flush_time = 0;
2341    page_cleaner->flush_pass = 0;
2342
2343    ulint lru_tm = 0;
2344    ulint list_tm = 0;
2345    ulint lru_pass = 0;
2346    ulint list_pass = 0;
```

Expected # of flushed pages from flush list

- buf/buf0flu.cc: page_cleaner_flush_pages_recommendation()

```
2348    for (ulint i = 0; i < page_cleaner->n_slots; i++) {
2349      page_cleaner_slot_t *slot;
2350
2351      slot = &page_cleaner->slots[i];
2352
2353      lru_tm += slot->flush_lru_time;
2354      lru_pass += slot->flush_lru_pass;
2355      list_tm += slot->flush_list_time;
2356      list_pass += slot->flush_list_pass;
2357
2358      slot->flush_lru_time = 0;
2359      slot->flush_lru_pass = 0;
2360      slot->flush_list_time = 0;
2361      slot->flush_list_pass = 0;
2362    }
2363
2364    mutex_exit(&page_cleaner->mutex);
```

Aggregate stats (time, pass) of all slots

Set all stats (time, pass) to 0 for next flushing

Expected # of flushed pages from flush list

- buf/buf0flu.cc: page_cleaner_flush_pages_recommendation()

```
2366      /* minimum values are 1, to avoid dividing by zero. */
2367      if (lru_tm < 1) {
2368        lru_tm = 1;
2369      }
2370      if (list_tm < 1) {
2371        list_tm = 1;
2372      }
2373      if (flush_tm < 1) {
2374        flush_tm = 1;
2375      }
2376
2377      if (lru_pass < 1) {
2378        lru_pass = 1;
2379      }
2380      if (list_pass < 1) {
2381        list_pass = 1;
2382      }
2383      if (flush_pass < 1) {
2384        flush_pass = 1;
2385      }
```

Set related monitor counters' value

```
2387      MONITOR_SET(MONITOR_FLUSH_ADAPTIVE_AVG_TIME_SLOT, list_tm / list_pass);
2388      MONITOR_SET(MONITOR_LRU_BATCH_FLUSH_AVG_TIME_SLOT, lru_tm / lru_pass);
2389
2390      MONITOR_SET(MONITOR_FLUSH_ADAPTIVE_AVG_TIME_THREAD,
2391                  list_tm / (srv_n_page_cleaners * flush_pass));
2392      MONITOR_SET(MONITOR_LRU_BATCH_FLUSH_AVG_TIME_THREAD,
2393                  lru_tm / (srv_n_page_cleaners * flush_pass));
2394      MONITOR_SET(MONITOR_FLUSH_ADAPTIVE_AVG_TIME_EST,
2395                  flush_tm * list_tm / flush_pass / (list_tm + lru_tm));
2396      MONITOR_SET(MONITOR_LRU_BATCH_FLUSH_AVG_TIME_EST,
2397                  flush_tm * lru_tm / flush_pass / (list_tm + lru_tm));
          MONITOR_SET(MONITOR_FLUSH_AVG_TIME, flush_tm / flush_pass);

          MONITOR_SET(MONITOR_FLUSH_ADAPTIVE_AVG_PASS,
                      list_pass / page_cleaner->n_slots);
2402      MONITOR_SET(MONITOR_LRU_BATCH_FLUSH_AVG_PASS,
2403                  lru_pass / page_cleaner->n_slots);
2404      MONITOR_SET(MONITOR_FLUSH_AVG_PASS, flush_pass);
2405
2406      prev_lsn = cur_lsn;
2407      prev_time = curr_time;
2408
2409      n_iterations = 0;
2410
2411      sum_pages = 0;
2412    }
```

Expected # of flushed pages from flush list

- buf/buf0flu.cc: page_cleaner_flush_pages_recommendation()

```
2414    oldest_lsn = buf_pool_get_oldest_modification_approx();
2415
2416    ut_ad(oldest_lsn <= log_get_lsn(*log_sys));
2417
2418    age = cur_lsn > oldest_lsn ? cur_lsn - oldest_lsn : 0;
2419
2420    pct_for_dirty = af_get_pct_fo
2421    pct_for_lsn = af_get_pct_for_
2422
2423    pct_total = ut_max(pct_for_di
2424
2425    /* Estimate pages to be flushed for the lsn progress */
2426    ulint sum_pages_for_lsn = 0;
2427    lsn_t target_lsn = oldest_lsn + lsn_avg_rate * buf_flush_lsn_scan_factor;
```

Get oldest LSN by traversing all the flush lists

Calculate **age**;
The difference between *current LSN* and *earliest LSN on the flush list*

Expected # of flushed pages from flush list

- buf/buf0flu.cc: page_cleaner_flush_pages_recommendation()

```
2414    oldest_lsn = buf_pool_get_oldest_modification_approx();
2415
2416    ut_ad(oldest_lsn <= log_get_lsn(*log_sys));
2417
2418    age = cur_lsn > oldest_lsn ? cur_lsn - oldest_lsn : 0;
2419
2420    pct_for_dirty = af_get_pct_for_dirty();
2421    pct_for_lsn = af_get_pct_for_lsn(age);
2422
2423    pct_total = ut_max(pct_for_dirty, pct_f
2424
2425    /* Estimate pages to be flushed for the
2426    ulint sum_pages_for_lsn = 0;
2427    lsn_t target_lsn = oldest_lsn + lsn_avg
```

Calculate percent of **io_capacity** to flush based on two factors:

① How many dirty pages there are in the buffer pool

② How quickly we are generating redo logs

Calculation of % of *io_capacity:* ①

- buf/buf0flu.cc: af_get_pct_for_dirty()

```
2212  /** Calculates if flushing is required based on number of dirty pages in
2213   the buffer pool.
2214   @return percent of io_capacity to flush to manage dirty page ratio */
2215  static ulint af_get_pct_for_dirty() {
2216    double dirty_pct = buf_get_modified_ratio_pct();
2217
2218    if (dirty_pct == 0.0) {
2219      /* No pages modified */
2220      return (0);
2221    }
2222
2223    ut_a(srv_max_dirty_pages_pct_lwm <= srv_max_buf_pool_modified_pct);
```

Get dirty page percentage ratio

Calculation of % of *io_capacity:* ①

- buf/buf0buf.cc: buf_get_modified_ratio_pct()

```
5443  /** Returns the ratio in percents of modified pages in the buffer pool /
5444   database pages in the buffer pool.
5445   @return modified page percentage ratio */
5446  double buf_get_modified_ratio_pct(void) {
5447    double ratio;
5448    ulint lru_len = 0;
5449    ulint free_len = 0;
5450    ulint flush_list_len = 0;
5451
5452    buf_get_total_list_len(&lru_len, &free_len, &flush_list_len);
5453
5454    ratio = static_cast<double>(100 * flush_list_len) / (1 + lru_len + free_len);
5455
5456    /* 1 + is there to avoid division by zero */
5457
5458    return (ratio);
5459  }
```

Calculation of % of *io_capacity:* ①

- buf/buf0flu.cc: af_get_pct_for_dirty()

```
2225    if (srv_max_dirty_pages_pct_lwm == 0) {
2226      /* The user has not set the option to preflush dirty
2227      pages as we approach the high water mark. */
2228      if (dirty_pct >= srv_max_buf_pool_modified_pct) {
2229        /* We have crossed the high water mark of dirty
2230        pages In this case we start flushing at 100% of
2231        innodb_io_capacity. */
2232        return (100);
2233      }
2234    } else if (dirty_pct >= srv_max_dirty_pages_pct_lwm) {
2235      /* We should start flushing pages gradually. */
2236      return (static_cast<ulint>((dirty_pct * 100) /
2237                                 (srv_max_buf_pool_modified_pct + 1)));
2238    }
2239
2240    return (0);
2241  }
```

$\geq high\ water\ mark(75)$

$\geq low\ water\ mark$

InnoDB tries to keep the ratio of dirty pages in the buffer pool smaller than *srv_max_buf_pool_modified_pct*(75.0)

Calculation of % of *io_capacity:* ②

- buf/buf0flu.cc: af_get_pct_for_lsn()

```
2243  /** Calculates if flushing is required based on redo generation rate.
2244   @return percent of io_capacity to flush to manage redo space */
2245  static ulint af_get_pct_for_lsn(lsn_t age) /*!< in: current age of LSN. */
2246  {
2247    lsn_t max_async_age;
2248    lsn_t lsn_age_factor;
2249    lsn_t af_lwm = (srv_adaptive_flushing_lwm * log_get_capacity()) / 100;
2250
2251    if (age < af_lwm) {
2252      /* No adaptive flushing. */
2253      return (0);
2254    }
2255
2256    max_async_age = log_get_max_modified_age_async();
```

Calculate low water mark of adaptive flushing based on *srv_adapative_flushing_lwm(10)* and log size

```
520     /** Capacity of log files excluding headers of the log files.
521     If the checkpoint age exceeds this, it is a serious error,
522     because it is possible we will then overwrite log and spoil
523     crash recovery. */
524     lsn_t lsn_capacity;
```

Calculation of % of *io_capacity:* ②

- buf/buf0flu.cc: af_get_pct_for_lsn()

```
2243  /** Calculates if flushing is required based on redo generation rate.
2244   @return percent of io_capacity to flush to manage redo space */
2245  static ulint af_get_pct_for_lsn(lsn_t age) /*!< in: current age of LSN. */
2246  {
2247    lsn_t max_async_age;
2248    lsn_t lsn_age_factor;
2249    lsn_t af_lwm = (srv_adaptive_flushing_lwm * log_get_capacity()) / 100;
2250
2251    if (age < af_lwm) {
2252      /* No adaptive flushing. */
2253      return (0);
2254    }
2255
2256    max_async_age = log_get_max_modified_age_async();
```

Maximum LSN difference; when exceeded, start asynchronous preflush

```
526    /** When the oldest dirty page age exceeds this value, we start
527    an asynchronous preflush of dirty pages. */
528    lsn_t max_modified_age_async;
```

Calculation of % of *io_capacity:* ②

- buf/buf0flu.cc: af_get_pct_for_lsn()

```
2258    if (age < max_async_age && !srv_adaptive_flushing) {
2259      /* We have still not reached the max_async point and
2260      the user has disabled adaptive flushing. */
2261      return (0);
2262    }
2263
2264    /* If we are here then we know that either:
2265    1) User has enabled adaptive flushing
2266    2) User may have disabled adaptive flushing but we have reached
2267    max_async_age. */
2268    lsn_age_factor = (age * 100) / max_async_age;
```

Percentage of *io_capacity* that should be used for flushing =

```
2271    return (static_cast<ulint>(((srv_max_io_capacity / srv_io_capacity) *
2272                                (lsn_age_factor * sqrt((double)lsn_age_factor))) /
2273                               7.5));
2274  }
```

Expected # of flushed pages from flush list

- buf/buf0flu.cc: page_cleaner_flush_pages_recommendation()

```
2414    oldest_lsn = buf_pool_get_oldest_modification_approx();
2415
2416    ut_ad(oldest_lsn <= log_get_lsn(*log_sys));
2417
2418    age = cur_lsn > oldest_lsn ? cur_lsn - oldest_lsn : 0;
2419
2420    pct_for_dirty = af_get_pct_for_dirty();
2421    pct_for_lsn = af_get_pct_for_lsn(age);
2422
2423    pct_total = ut_max(pct_for_dirty, pct_for_lsn);
2424
2425    /* Estimate pages to be flushed for the lsn progress */
2426    ulint sum_pages_for_lsn = 0;
2427    lsn_t target_lsn = oldest_lsn + lsn_avg_rate * buf_flush_lsn_scan_factor;
```

```
92   /** Factor for scan length to determine n_pages for intended oldest LSN
93   progress */
94   static ulint buf_flush_lsn_scan_factor = 3;
```

Expected # of flushed pages from flush list

- buf/buf0flu.cc: page_cleaner_flush_pages_recommendation()

```
2429  for (ulint i = 0; i < srv_buf_pool_instances; i++) {
2430    buf_pool_t *buf_pool = buf_pool_from_array(i);
2431    ulint pages_for_lsn = 0;
2432
2433    buf_flush_list_mutex_enter(buf_pool);
2434    for (buf_page_t *b = UT_LIST_GET_LAST(buf_pool->flush_list); b != NULL;
2435         b = UT_LIST_GET_PREV(list, b)) {
2436      if (b->oldest_modification > target_lsn) {
2437        break;
2438      }
2439      ++pages_for_lsn;
2440    }
2441    buf_flush_list_mutex_exit(buf_pool);
2442
2443    sum_pages_for_lsn += pages_for_lsn;
```

For every page in flush list,
counts the number of pages for which
LSN of the page is less than target LSN.

Expected # of flushed pages from flush list

- buf/buf0flu.cc: page_cleaner_flush_pages_recommendation()

```
2445      mutex_enter(&page_cleaner->mutex);
2446      ut_ad(page_cleaner->slots[i].state == PAGE_CLEANER_STATE_NONE);
2447      page_cleaner->slots[i].n_pages_requested =
2448          pages_for_lsn / buf_flush_lsn_scan_factor + 1;
2449      mutex_exit(&page_cleaner->mutex);
2450    }
2451
2452    sum_pages_for_lsn /= buf_flush_lsn_scan_factor;
2453    if (sum_pages_for_lsn < 1) {
2454      sum_pages_for_lsn = 1;
2455    }
2456
2457    /* Cap the maximum IO capacity that we are going to use by
2458    max_io_capacity. Limit the value to avoid too quick increase */
2459    ulint pages_for_lsn =
2460        std::min<ulint>(sum_pages_for_lsn, srv_max_io_capacity * 2);
```

Cap the maximum I/O capacity

Expected # of flushed pages from flush list

- buf/buf0flu.cc: page_cleaner_flush_pages_recommendation()

```
2462    n_pages = (PCT_IO(pct_total) + avg_page_rate + pages_for_lsn) / 3;
2463
2464    if (n_pages > srv_max_io_capacity) {
2465      n_pages = srv_max_io_capacity;
2466    }
2467
2468    /* Normalize request for each instance */
2469    mutex_enter(&page_cleaner->mutex);
2470    ut_ad(page_cleaner->n_slots_requested == 0);
2471    ut_ad(page_cleaner->n_slots_flushing == 0);
2472    ut_ad(page_cleaner->n_slots_finished == 0);
```

Expected # of flushed pages from flush list

- buf/buf0flu.cc: page_cleaner_flush_pages_recommendation()

```
2474  for (ulint i = 0; i < srv_buf_pool_instances; i++) {
2475    /* if REDO has enough of free space,
2476    don't care about age distribution of pages */
2477    page_cleaner->slots[i].n_pages_requested =
2478        pct_for_lsn > 30 ? page_cleaner->slots[i].n_pages_requested * n_pages /
2479                                   sum_pages_for_lsn +
2480                               1
2481                         : n_pages / srv_buf_pool_instances;
2482  }
2483  mutex_exit(&page_cleaner->mutex);
2484
2485  MONITOR_SET(MONITOR_FLUSH_N_TO_FLUSH_REQUESTED, n_pages);
2486
2487  MONITOR_SET(MONITOR_FLUSH_N_TO_FLUSH_BY_AGE, sum_pages_for_lsn);
```

Expected # of flushed pages from flush list

- buf/buf0flu.cc: page_cleaner_flush_pages_recommendation()

```
2489    MONITOR_SET(MONITOR_FLUSH_AVG_PAGE_RATE, avg_page_rate);
2490    MONITOR_SET(MONITOR_FLUSH_LSN_AVG_RATE, lsn_avg_rate);
2491    MONITOR_SET(MONITOR_FLUSH_PCT_FOR_DIRTY, pct_for_dirty);
2492    MONITOR_SET(MONITOR_FLUSH_PCT_FOR_LSN, pct_for_lsn);
2493
2494    *lsn_limit = LSN_MAX;
2495
2496    return (n_pages);
2497  }
```

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: buf_flush_page_coordinator_thread()

```
3051        } else if (srv_check_activity(last_activity)) {
3052          ulint n_to_flush;
3053          lsn_t lsn_limit = 0;
3054
3055          /* Estimate pages from flush_list to be flushed */
3056          if (ret_sleep == OS_SYNC_TIME_EXCEEDED) {
3057            last_activity = srv_get_activity_count();
3058            n_to_flush =
3059                page_cleaner_flush_pages_recommendation(&lsn_limit, last_pages);
3060          } else {
3061            n_to_flush = 0;
3062          }
3063
3064          /* Request flushing for threads */
3065          pc_request(n_to_flush, lsn_limit);
3066
3067          ulint tm = ut_time_ms();
```

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: pc_request()

```
2575  /**
2576  Requests for all slots to flush all buffer pool instances.
2577  @param min_n    wished minimum mumber of blocks flushed
2578                  (it is not guaranteed that the actual number is that big)
2579  @param lsn_limit in the case BUF_FLUSH_LIST all blocks whose
2580                  oldest_modification is smaller than this should be flushed
2581                  (if their number does not exceed min_n), otherwise ignored
2582  */
2583  static void pc_request(ulint min_n, lsn_t lsn_limit) {
2584    if (min_n != ULINT_MAX) {
2585      /* Ensure that flushing is spread evenly amongst the
2586      buffer pool instances. When min_n is ULINT_MAX
2587      we need to flush everything up to the lsn limit
2588      so no limit here. */
2589      min_n = (min_n + srv_buf_pool_instances - 1) / srv_buf_pool_instances;
2590    }
2591
2592     mutex_enter(&page_cleaner->mutex);
```

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: pc_request()

```
2598    page_cleaner->requested = (min_n > 0);
2599    page_cleaner->lsn_limit = lsn_limit;
2600
2601    for (ulint i = 0; i < page_cleaner->n_slots; i++) {
2602      page_cleaner_slot_t *slot = &page_cleaner->slots[i];
2603
2604      ut_ad(slot->state == PAGE_CLEANER_STATE_NONE);
2605
2606      if (min_n == ULINT_MAX) {
2607        slot->n_pages_requested = ULINT_MAX;
2608      } else if (min_n == 0) {
2609        slot->n_pages_requested = 0;
2610      }
2611
2612      /* slot->n_pages_requested was already set by
2613      page_cleaner_flush_pages_recommendation() */
2614
2615      slot->state = PAGE_CLEANER_STATE_REQUESTED;
2616    }
```

For all *page_cleaner_slots*

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: pc_request()

```
2618    page_cleaner->n_slots_requested = page_cleaner->n_slots;
2619    page_cleaner->n_slots_flushing = 0;
2620    page_cleaner->n_slots_finished = 0;
2621
2622    os_event_set(page_cleaner->is_requested);
2623
2624    mutex_exit(&page_cleaner->mutex);
2625  }
```

Requests for all slots to flush all buffer pool instances

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: buf_flush_page_coordinator_thread()

```
3069        /* Coordinator also treats requests */
3070        while (pc_flush_slot() > 0) {
3071          /* No op */
3072        }
3073
3074        /* only coordinator is using these counters,
3075        so no need to protect by lock. */
3076        page_cleaner->flush_time += ut_time_ms() - tm;
3077        page_cleaner->flush_pass++;
3078
3079        /* Wait for all slots to be finished */
3080        ulint n_flushed_lru = 0;
3081        ulint n_flushed_list = 0;
3082
3083        pc_wait_finished(&n_flushed_lru, &n_flushed_list);
3084
3085        if (n_flushed_list > 0 || n_flushed_lru > 0) {
3086          buf_flush_stats(n_flushed_list, n_flushed_lru);
3087        }
```

Do flush for each slot

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: pc_flush_slot()

```
2627  /**
2628  Do flush for one slot.
2629  @return the number of the slots which has not been treated yet. */
2630  static ulint pc_flush_slot(void) {
2631    ulint lru_tm = 0;
2632    ulint list_tm = 0;
2633    int lru_pass = 0;
2634    int list_pass = 0;
2635
2636    mutex_enter(&page_cleaner->mutex);
2637
2638    if (page_cleaner->n_slots_requested > 0) {
2639      page_cleaner_slot_t *slot = NULL;
2640      ulint i;
```

If there is at least one REQUESTED slot,

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: pc_flush_slot()

```
2642        for (i = 0; i < page_cleaner->n_slots; i++) {
2643          slot = &page_cleaner->slots[i];
2644
2645          if (slot->state == PAGE_CLEANER_STATE_REQUESTED) {
2646            break;
2647          }
2648        }
2649
2650        /* slot should be found because
2651        page_cleaner->n_slots_requested > 0 */
2652        ut_a(i < page_cleaner->n_slots);
2653
2654        buf_pool_t *buf_pool = buf_pool_from_array(i);
2655
2656        page_cleaner->n_slots_requested--;
2657        page_cleaner->n_slots_flushing++;
2658        slot->state = PAGE_CLEANER_STATE_FLUSHING;
```

Find the REQUESTED slot

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: pc_flush_slot()

```
2660    if (page_cleaner->n_slots_requested == 0) {
2661      os_event_reset(page_cleaner->is_requested);
2662    }
2663
2664    if (!page_cleaner->is_running) {
2665      slot->n_flushed_lru = 0;
2666      slot->n_flushed_list = 0;
2667      goto finish_mutex;
2668    }
2669
2670    mutex_exit(&page_cleaner->mutex);
2671
2672    lru_tm = ut_time_ms();
2673
2674    /* Flush pages from end of LRU if required */
2675    slot->n_flushed_lru = buf_flush_LRU_list(buf_pool);
2676
2677    lru_tm = ut_time_ms() - lru_tm;
2678    lru_pass++;
```

First, do LRU flushing

LRU LIST FLUSHING

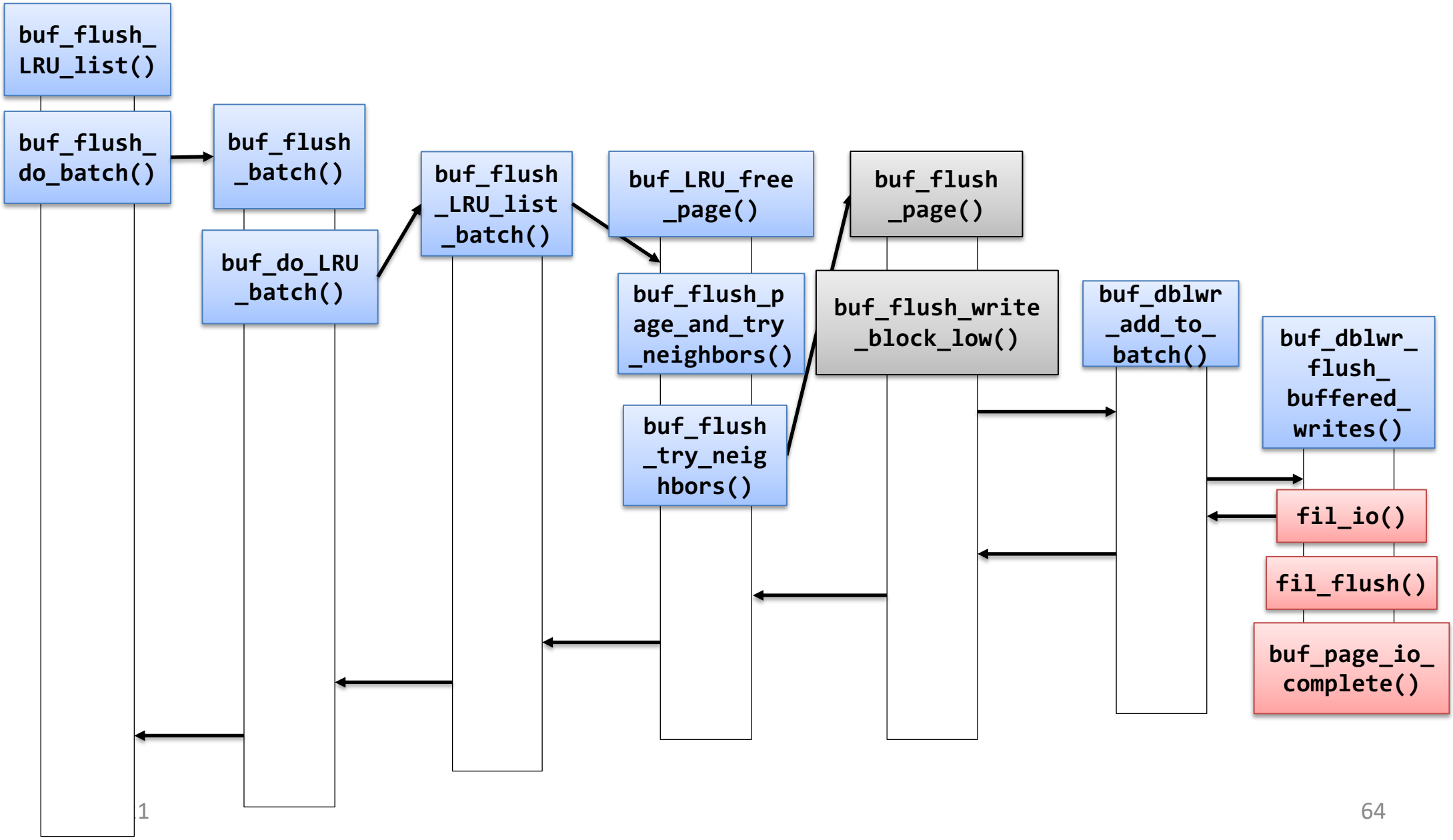
LRU List Flushing

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

- 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

- *innodb_LRU_scan_depth* = 1024 /* default */
  - How **deeply** to examine tail for dirty pages
  - User thread may scan up to this depth as well if no page available in free list
  - Important to tune to prevent synchronous flushes (= spf)

LRU List Flushing



LRU List Flushing

- buf/buf0flu.cc: buf_flush_LRU_list()

```
2159  /**
2160  Clears up tail of the LRU list of a given buffer pool instance:
2161  * Put replaceable pages at the tail of LRU to the free list
2162  * Flush dirty pages at the tail of LRU to the disk
2163  The depth to which we scan each buffer pool is controlled by dynamic
2164  config parameter innodb_LRU_scan_depth.
2165  @param buf_pool buffer pool instance
2166  @return total pages flushed */
2167  static ulint buf_flush_LRU_list(buf_pool_t *buf_pool) {
2168    ulint scan_depth, withdraw_depth;
2169    ulint n_flushed = 0;
2170
2171    ut_ad(buf_pool);
2172
2173    /* srv_LRU_scan_depth can be arbitrarily large value.
2174    We cap it with current LRU size. */
2175    scan_depth = UT_LIST_GET_LEN(buf_pool->LRU);
2176    withdraw_depth = buf_get_withdraw_depth(buf_pool);
```

LRU List Flushing

- buf/buf0flu.cc: buf_flush_LRU_list()

```
2178    if (withdraw_depth > srv_LRU_scan_depth) {
2179      scan_depth = ut_min(withdraw_depth, scan_depth);
2180    } else {
2181      scan_depth = ut_min(static_cast<ulint>(srv_LRU_scan_depth), scan_depth);
2182    }
2183
2184    /* Currently one of page_cleaners is the only thread
2185    that can trigger an LRU flush at the same time.
2186    So, it is not possible that a batch triggered during
2187    last iteration is still running, */
2188    buf_flush_do_batch(buf_pool, BUF_FLUSH_LRU, scan_depth, 0, &n_flushed);
2189
2190    return (n_flushed);
2191  }
```

Batch LRU flush

LRU List Flushing

- buf/buf0flu.cc: buf_flush_do_batch()

```
1937  bool buf_flush_do_batch(buf_pool_t *buf_pool, buf_flush_t type, ulint min_n,
1938                          lsn_t lsn_limit, ulint *n_processed) {
                                    ...
1945    if (!buf_flush_start(buf_pool, type)) {
1946      return (false);
1947    }
1948
1949    ulint page_count = buf_flush_batch(buf_pool, type, min_n, lsn_limit);
1950
1951    buf_flush_end(buf_pool, type);
1952
1953    if (n_processed != NULL) {
1954      *n_processed = page_count;
1955    }
1956
1957    return (true);
1958  }
```

LRU List Flushing

- buf/buf0flu.cc: buf_flush_batch()

```
1799  static ulint buf_flush_batch(buf_pool_t *buf_pool, buf_flush_t flush_type,
1800                               ulint min_n, lsn_t lsn_limit) {
                                   ...
1813    /* Note: The buffer pool mutexes is released and reacquired within
1814    the flush functions. */
1815    switch (flush_type) {
1816      case BUF_FLUSH_LRU:
1817        mutex_enter(&buf_pool->LRU_list_mutex);
1818        count = buf_do_LRU_batch(buf_pool, min_n);
1819        mutex_exit(&buf_pool->LRU_list_mutex);
1820        break;
1821      case BUF_FLUSH_LIST:
1822        count = buf_do_flush_list_batch(buf_pool, min_n, lsn_limit);
1823        break;
1824      default:
1825        ut_error;
1826    }
```

Do LRU batch

LRU List Flushing

- buf/buf0flu.cc: buf_do_LRU_batch()

```
1703  static ulint buf_do_LRU_batch(buf_pool_t *buf_pool, ulint max) {
1704    ulint count = 0;
1705
1706    ut_ad(mutex_own(&buf_pool->LRU_list_mutex));
1707
1708    if (buf_LRU_evict_from_unzip_LRU(buf_pool)) {
1709      count += buf_free_from_unzip_LRU_list_batch(buf_pool, max);
1710    }
1711
1712    if (max > count) {
1713      count += buf_flush_LRU_list_batch(buf_pool, max - count);
1714    }
1715
1716    return (count);
1717  }
```

LRU List Flushing

- buf/buf0flu.cc: buf_flush_LRU_list_batch()

```
1616  static ulint buf_flush_LRU_list_batch(buf_pool_t *buf_pool, ulint max) {
1617    buf_page_t *bpage;
1618    ulint scanned = 0;
1619    ulint evict_count = 0;
1620    ulint count = 0;
1621    ulint free_len = UT_LIST_GET_LEN(buf_pool->free);
1622    ulint lru_len = UT_LIST_GET_LEN(buf_pool->LRU);
1623    ulint withdraw_depth;
1624
1625    ut_ad(mutex_own(&buf_pool->LRU_list_mutex));
1626
1627    withdraw_depth = buf_get_withdraw_depth(buf_pool);
1628
1629    for (bpage = UT_LIST_GET_LAST(buf_pool->LRU);
1630         bpage != NULL && count + evict_count < max &&
1631         free_len < srv_LRU_scan_depth + withdraw_depth &&
1632         lru_len > BUF_LRU_MIN_LEN;
1633         ++scanned, bpage = buf_pool->lru_hp.get()) {
```

Start flushing with the last page from LRU

LRU List Flushing

- buf/buf0flu.cc: buf_flush_LRU_list_batch()

```
1634        buf_page_t *prev = UT_LIST_GET_PREV(LRU, bpage);
1635        buf_pool->lru_hp.set(prev);
1636
1637        BPageMutex *block_mutex = buf_page_get_mutex(bpage);
1638
1639        bool acquired = mutex_enter_nowait(block_mutex) == 0;
1640
1641        if (acquired && buf_flush_ready_for_replace(bpage)) {
1642          /* block is ready for eviction i.e., it is
1643          clean and is not IO-fixed or buffer fixed. */
1644          if (buf_LRU_free_page(bpage, true)) {
1645            ++evict_count;
1646            mutex_enter(&buf_pool->LRU_list_mutex);
1647          } else {
1648            mutex_exit(block_mutex);
1649          }
```

LRU List Flushing

- buf/buf0flu.cc: buf_flush_ready_for_replace()

```
587  ibool buf_flush_ready_for_replace(buf_page_t *bpage) {
588  #ifdef UNIV_DEBUG
589    buf_pool_t *buf_pool = buf_pool_from_bpage(bpage);
590    ut_ad(mutex_own(&buf_pool->LRU_list_mutex));
591  #endif /* UNIV_DEBUG */
592    ut_ad(mutex_own(buf_page_get_mutex(bpage)));
593    ut_ad(bpage->in_LRU_list);
594
595    if (buf_page_in_file(bpage)) {
596      return (bpage->oldest_modification == 0 && bpage->buf_fix_count == 0 &&
597              buf_page_get_io_fix(bpage) == BUF_IO_NONE);
598    }
599
600    ib::fatal(ER_IB_MSG_123) << "Buffer block " << bpage << " state "
601                             << bpage->state << " in the LRU list!";
602
603    return (FALSE);
604  }
```

Check whether the given page is **clean** or not

LRU List Flushing

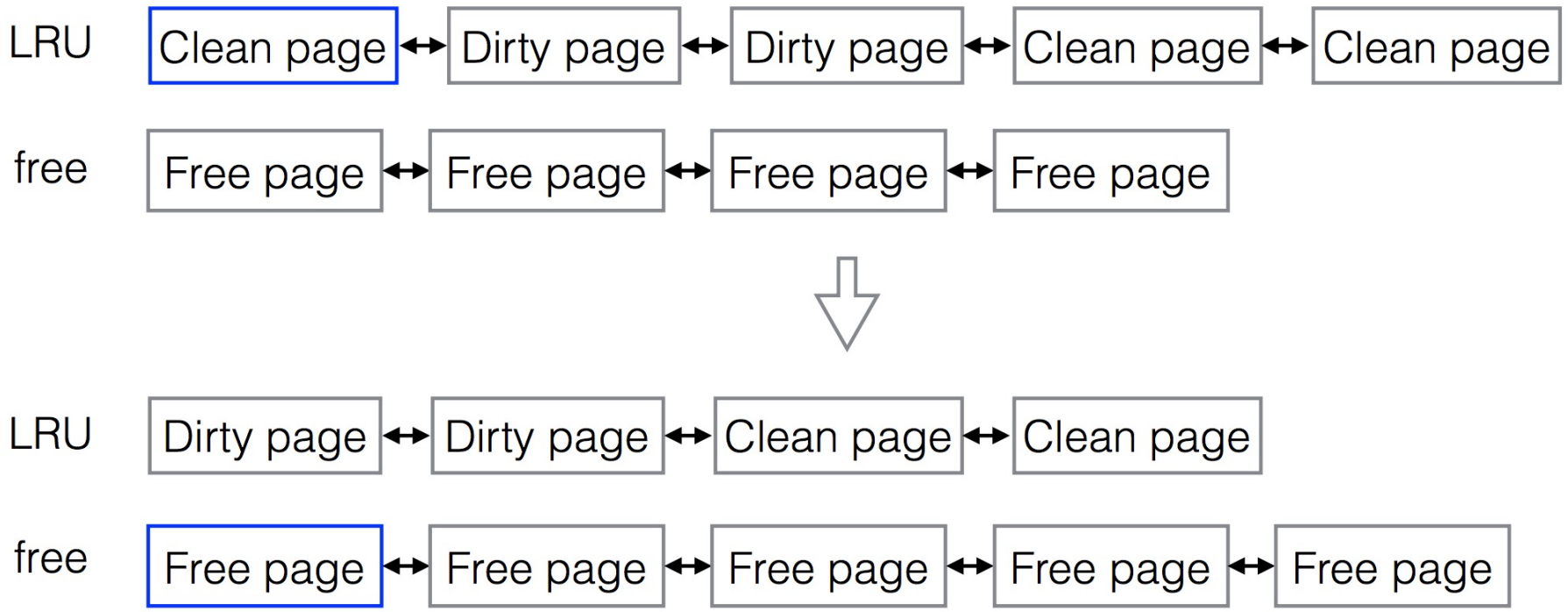
- buf/buf0flu.cc: buf_flush_LRU_list_batch()

```
1634        buf_page_t *prev = UT_LIST_GET_PREV(LRU, bpage);
1635        buf_pool->lru_hp.set(prev);
1636
1637        BPageMutex *block_mutex = buf_page_get_mutex(bpage);
1638
1639        bool acquired = mutex_enter_nowait(block_mutex) == 0;
1640
1641        if (acquired && buf_flush_ready_for_replace(bpage)) {
1642          /* block is ready for eviction i.e., it is
1643          clean and is not IO-fixed or buffer fixed. */
1644          if (buf_LRU_free_page(bpage, true)) {
1645            ++evict_count;
1646            mutex_enter(&buf_pool->LRU_list_mutex);
1647          } else {
1648            mutex_ex
1649          }
```

It there is any replaceable page, free the page
(LRU list ➔ free list)

LRU List Flushing

- Evicting/flushing pages from the **LRU list** and putting them on the **free list**



LRU List Flushing

- buf/buf0flu.cc: buf_flush_LRU_list_batch()

```
1650        } else if (acquired && buf_flush_ready_for_flush(bpage, BUF_FLUSH_LRU)) {
1651          /* Block is ready for flush. Dispatch an IO
1652          request. The IO helper thread will put it on
1653          free list in IO completion routine. */
1654          mutex_exit(block_mutex);
1655          buf_flush_page_and_try_neighbors(bpage, BUF_FLUSH_LRU, max, &count);
1656        } else if (!acquired) {
1657          ut_ad(buf_pool->lru_hp.is_hp(prev));
1658        } else {
1659          /* Can't evict or dispatch this block. Go to
1660          previous. */
1661          mutex_exit(block_mutex);
1662          ut_ad(buf_pool->lru_hp.is_hp(prev));
1663        }
```

Else if,

LRU List Flushing

- buf/buf0flu.cc: buf_flush_ready_for_flush()

```
610  bool buf_flush_ready_for_flush(buf_page_t *bpage, buf_flush_t flush_type) {
                                        ...
625    if (bpage->oldest_modification == 0 ||
626        buf_page_get_io_fix_unlocked(bpage) != BUF_IO_NONE) {
627      return (false);
628    }
629
630    ut_ad(bpage->i
631
632    switch (flush_type) {
633      case BUF_FLUSH_LIST:
634        return (buf_page_get_state(bpage) != BUF_BLOCK_REMOVE_HASH);
635      case BUF_FLUSH_LRU:
636      case BUF_FLUSH_SINGLE_PAGE:
637        return (true);
```

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

LRU List Flushing

- buf/buf0flu.cc: buf_flush_LRU_list_batch()

```
1650        } else if (acquired && buf_flush_ready_for_flush(bpage, BUF_FLUSH_LRU)) {
1651          /* Block is ready for flush. Dispatch an IO
1652          request. The IO helper thread will put it on
1653          free list in IO completion routine. */
1654          mutex_exit(block_mutex);
1655          buf_flush_page_and_try_neighbors(bpage, BUF_FLUSH_LRU, max, &count);
1656        } else if (!acquired) {
1657          ut_ad(buf_pool->lru_hp.is_hp(prev));
1658        } else {
1659          /* Can't evict or dispatch this block. Go to
1660          previous. */
1661          mutex_exit(block_mutex);
1662          ut_ad(buf_pool->lru_hp.is_hp(prev));
1663        }
```

If the given page is ready for flush, try to flush with neighbor pages

LRU List Flushing

- buf/buf0flu.cc: buf_flush_try_neighbors()

```
1330  static ulint buf_flush_try_neighbors(const page_id_t &page_id,
1331                                       buf_flush_t flush_type, ulint n_flushed,
1332                                       ulint n_to_flush) {
                                     ...
1402    for (i = low; i < high; i++) {
                                     ...
1443      if (flush_type != BUF_FLUSH_LRU || i == page_id.page_no() ||
1444          buf_page_is_old(bpage)) {
1445        if (buf_flush_ready_for_flush(bpage, flush_type) &&
1446            (i == page_id.page_no() || bpage->buf_fix_count == 0)) {
1447          /* We also try to flush those
1448          neighbors != offset */
1449
1450          if (buf_flush_page(buf_pool, bpage, flush_type, false)) {
1451            ++count;
1452          } else {
1453            mutex_exit(block_mutex);
1454          }
```

For all flushable pages within the flush area

Flush page, but no sync

LRU List Flushing

- buf/buf0flu.cc: buf_flush_page()

```
1129  ibool buf_flush_page(buf_pool_t *buf_pool, buf_page_t *bpage,
1130                       buf_flush_t flush_type, bool sync) {
                                    ...
1157    rw_lock_t *rw_lock = NULL;
                                    ...
1172      rw_lock = &reinterpret_cast<buf_block_t *>(bpage)->lock;
1173      if (flush_type != BUF_FLUSH_LIST) {
1174        flush = rw_lock_sx_lock_nowait(rw_lock, BUF_IO_WRITE);
                                    ...
1181    if (flush) {
1182      /* We are committed to flushing by the time we get here */
1183
1184      mutex_enter(&buf_pool->flush_state_mutex);
1185
1186      buf_page_set_io_fix(bpage, BUF_IO_WRITE);
1187
1188      buf_page_set_flush_type(bpage, flush_type);
```

Get lock

Set fix and flush type

LRU List Flushing

- buf/buf0flu.cc: buf_flush_page()

```
1200        if (!fsp_is_system_temporary(bpage->id.space()) &&
1201            buf_pool->track_page_lsn != LSN_MAX) {
1202          page_t *frame;
1203          lsn_t frame_lsn;
1204
1205          frame = bpage->zip.data;
1206
1207          if (!frame) {
1208            frame = ((buf_block_t *)bpage)->frame;
1209          }
1210          frame_lsn = mach_read_from_8(frame + FIL_PAGE_LSN);
1211
1212          arch_page_sys->track_page(bpage, buf_pool->track_page_lsn, frame_lsn,
1213                                    false);
1214        }
```

Get the page frame

LRU List Flushing

- buf/buf0flu.cc: buf_flush_page()

```
1216        mutex_exit(&buf_pool->flush_state_mutex);
1217
1218        mutex_exit(block_mutex);
                                    ...
1247        /* Even though bpage is not protected by any mutex at this
1248        point, it is safe to access bpage, because it is io_fixed and
1249        oldest_modification != 0.  Thus, it cannot be relocated in the
1250        buffer pool or removed from flush_list or LRU_list. */
1251
1252        buf_flush_write_block_low(bpage, flush_type, sync);
1253    }
```

Writes a flushable page from the buffer pool to a file

LRU List Flushing

- buf/buf0flu.cc: buf_flush_write_block_low()

```
1003  static void buf_flush_write_block_low(buf_page_t *bpage, buf_flush_t flush_type,
1004                                        bool sync) {

                                    ...

1033    /* Force the log to the disk before writing the modified block */
1034    if (!srv_read_only_mode) {
1035      Wait_stats wait_stats;
1036
1037      wait_stats = log_write_up_to(*log_sys, bpage->newest_modification, true);
1038
1039      MONITOR_INC_WAIT_STATS_EX(MONITOR_ON_LOG_, _PAGE_WRITTEN, wait_stats);
1040    }
```

Flush log (transaction log – WAL)

LRU List Flushing

- buf/buf0flu.cc: buf_flush_write_block_low()

```
1062        case BUF_BLOCK_FILE_PAGE:
1063          frame = bpage->zip.data;
1064          if (!frame) {
1065            frame = ((buf_block_t *)bpage)->frame;
1066          }
1067
1068          buf_flush_init_for_writing(
1069              reinterpret_cast<const buf_block_t *>(bpage),
1070              reinterpret_cast<const buf_block_t *>(bpage)->frame,
1071              bpage->zip.data ? &bpage->zip : NULL, bpage->newest_modification,
1072              fsp_is_checksum_disabled(bpage->id.space()));
1073          break;
1074      }
```

LRU List Flushing

- buf/buf0flu.cc: buf_flush_write_block_low()

Doublewrite off case

```
1080    if (!srv_use_doublewrite_buf || buf_dblwr == NULL || srv_read_only_mode ||
1081        fsp_is_system_temporary(bpage->id.space())) {
1082      ut_ad(!srv_read_only_mode || fsp_is_system_temporary(bpage->id.space()));
1083
1084      ulint type = IORequest::WRITE | IORequest::DO_NOT_WAKE;
1085
1086      dberr_t err;
1087      IORequest request(type);
1088
1089      err = fil_io(request, sync, bpage->id, bpage->size, 0,
1090                   bpage->size.physical(), frame, bpage);
1091
1092      ut_a(err == DB_SUCCESS);
```

LRU List Flushing

- buf/buf0flu.cc: buf_flush_write_block_low()

```
1094    } else if (flush_type == BUF_FLUSH_SINGLE_PAGE) {
1095      buf_dblwr_write_single_page(bpage, sync);
1096    } else {
1097      ut_ad(!sync);
1098      buf_dblwr_add_to_batch(bpage);
1099    }
1100
1101    /* When doing single page flushing the IO is done synchronously
1102    and we flush the changes to disk only for the tablespace we
1103    are working on. */
1104    if (sync) {
1105      ut_ad(flush_type == BUF_FLUSH_SINGLE_PAGE);
1106      fil_flush(bpage->id.space());
1107
1108      /* true means we want to evict this page from the
1109      LRU list as well. */
1110      buf_page_io_complete(bpage, true);
1111    }
```

Add the page to the doublewrite buffer

LRU List Flushing

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

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

LRU List Flushing

- buf/buf0flu.cc: buf_flush_do_batch()

```
1937  bool buf_flush_do_batch(buf_pool_t *buf_pool, buf_flush_t type, ulint min_n,
1938                          lsn_t lsn_limit, ulint *n_processed) {
                                    ...
1945    if (!buf_flush_start(buf_pool, type)) {
1946      return (false);
1947    }
1948
1949    ulint page_count = buf_flush_batch(buf_pool, type, min_n, lsn_limit);
1950
1951    buf_flush_end(buf_pool, type);
1952
1953    if (n_processed != NULL) {
1954      *n_processed = page_count;
1955    }
1956
1957    return (true);
1958  }
```

LRU List Flushing

- buf/buf0flu.cc: buf_flush_end()

```
1876  static void buf_flush_end(buf_pool_t *buf_pool, buf_flush_t flush_type) {
1877    mutex_enter(&buf_pool->flush_state_mutex);
1878
1879    buf_pool->init_flush[flush_type] = FALSE;
1880
1881    buf_pool->try_LRU_scan = TRUE;
1882
1883    if (buf_pool->n_flush[flush_type] == 0) {
1884      /* The running flush batch has ended */
1885
1886      os_event_set(buf_pool->no_flush[flush_type]);
1887    }
1888
1889    mutex_exit(&buf_pool->flush_state_mute
1890
1891    if (!srv_read_only_mode) {
1892      buf_dblwr_flush_buffered_writes();
```

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

; See this later

LRU List Flushing: Complete I/O

- After all the work for flushing is complete, the following function is called last to complete I/O
- buf/buf0buf.cc: buf_page_io_complete()

```
4657  bool buf_page_io_complete(buf_page_t *bpage, bool evict) {
                                        ...
4867      case BUF_IO_WRITE:
4868        /* Write means a flush operation: call the completion
4869        routine in the flush system */
4870
4871        buf_flush_write_complete(bpage);
```

LRU List Flushing: Complete I/O

- buf/buf0buf.cc: buf_page_io_complete()

```
4879        /* We decide whether or not to evict the page from the
4880        LRU list based on the flush_type.
4881        * BUF_FLUSH_LIST: don't evict
4882        * BUF_FLUSH_LRU: always evict
4883        * BUF_FLUSH_SINGLE_PAGE: eviction preference is passed
4884        by the caller explicitly. */
4885        if (buf_page_get_flush_type(bpage) == BUF_FLUSH_LRU) {
4886          evict = true;
4887          ut_ad(have_LRU_mutex);
4888        }
4889
4890        if (evict && buf_LRU_free_page(bpage, true)) {
4891          have_LRU_mutex = false;
4892        } else {
4893          mutex_exit(buf_page_get_mutex
4894        }
4895        if (have_LRU_mutex) {
4896          mutex_exit(&buf_pool->LRU_list_mutex);
4897        }
```

free the page ***(LRU list ➔ free list)***

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: pc_flush_slot()

```
2685        /* Flush pages from flush_list if required */
2686        if (page_cleaner->requested) {
2687          list_tm = ut_time_ms();
2688
2689          slot->succeeded_list =
2690              buf_flush_do_batch(buf_pool, BUF_FLUSH_LIST, slot->n_pages_requested,
2691                                 page_cleaner->lsn_limit, &slot->n_flushed_list);
2692
2693          list_tm = ut_time_ms() - list_tm;
2694          list_pass++;
2695        } else {
2696          slot->n_flushed_list = 0;
2697          slot->succeeded_list = true;
2698        }
```

Second, do flush list flushing

FLUSH LIST FLUSHING

Flush List Flushing

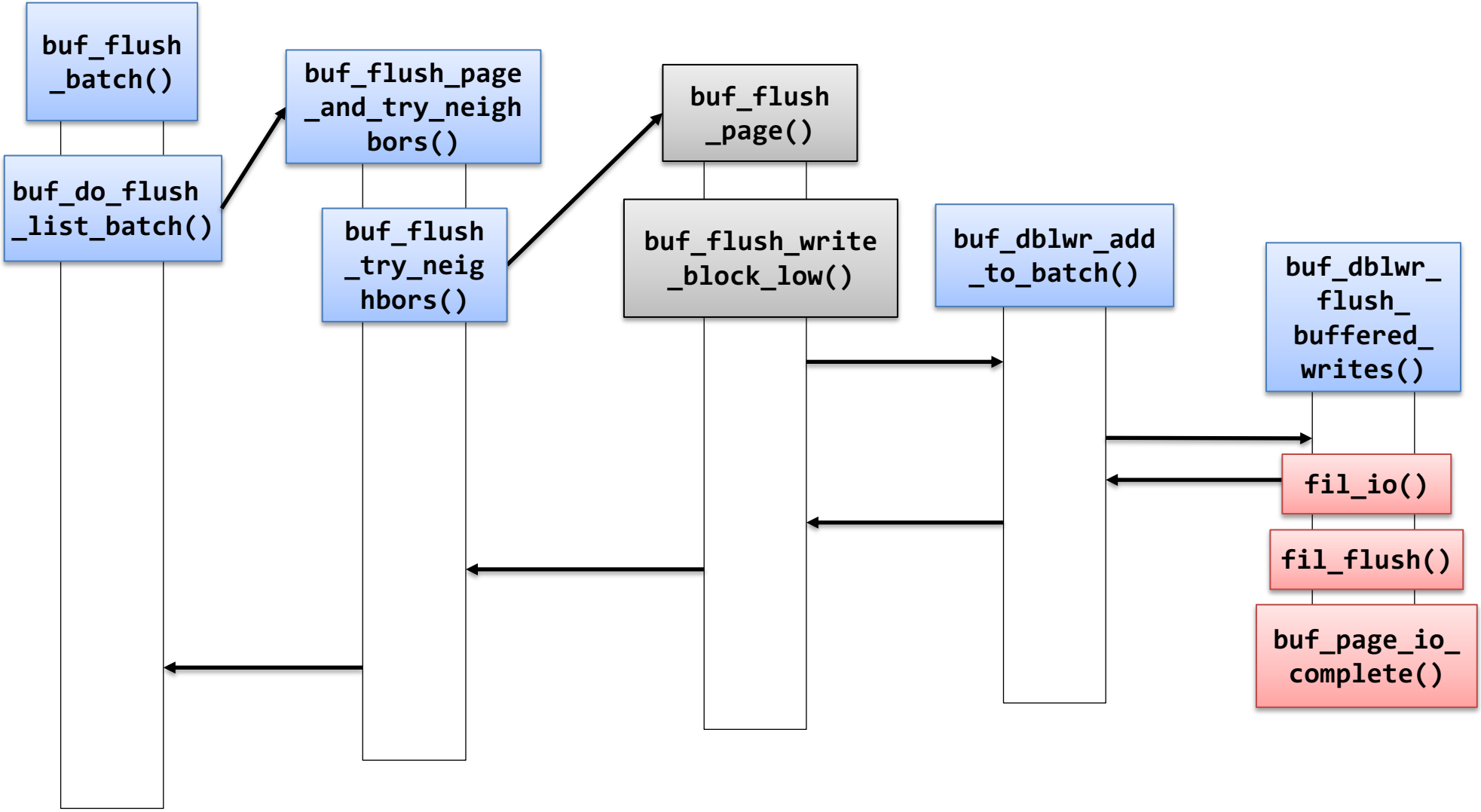
- buf/buf0flu.cc: *buf_flush_do_flush_list_batch()*
- Flushing to advance “earliest modify LSN”
  - To free log space so it can be reduced
  - Flush list size is capped by the **redo log size**
- Pages are moved from flush list when changes have been synced to disk
- Number of pages to flush per cycle depends on the load

Flush List Flushing

- *innodb_io_capacity* = 200 /* default */
  - Limit on rate flushing pages during *idle* time, or during *shutdown*
  - Change buffer merges at a rate of 5-55% of *innodb_io_capacity*

- ***innodb_io_capacity_max*** = 2000 /* default */
  - Limit on rate of flushing during busy time

Flush List Flushing



Flush List Flushing

- buf/buf0flu.cc: buf_flush_do_batch()

```
1937  bool buf_flush_do_batch(buf_pool_t *buf_pool, buf_flush_t type, ulint min_n,
1938                          lsn_t lsn_limit, ulint *n_processed) {
                                    ...
1945    if (!buf_flush_start(buf_pool, type)) {
1946      return (false);
1947    }
1948
1949    ulint page_count = buf_flush_batch(buf_pool, type, min_n, lsn_limit);
1950
1951    buf_flush_end(buf_pool, type);
1952
1953    if (n_processed != NULL) {
1954      *n_processed = page_count;
1955    }
1956
1957    return (true);
1958  }
```

Flush List Flushing

- buf/buf0flu.cc: buf_flush_batch()

```
1799  static ulint buf_flush_batch(buf_pool_t *buf_pool, buf_flush_t flush_type,
1800                               ulint min_n, lsn_t lsn_limit) {
                                   ...
1813    /* Note: The buffer pool mutexes is released and reacquired within
1814    the flush functions. */
1815    switch (flush_type) {
1816      case BUF_FLUSH_LRU:
1817        mutex_enter(&buf_pool->LRU_list_mutex);
1818        count = buf_do_LRU_batch(buf_pool, min_n);
1819        mutex_exit(&buf_pool->LRU_list_mutex);
1820        break;
1821      case BUF_FLUSH_LIST:
1822        count = buf_do_flush_list_batch(buf_pool, min_n, lsn_limit);
1823        break;
1824      default:
1825        ut_error;
1826    }
```

Do flush list flushing

Flush List Flushing

- buf/buf0flu.cc: buf_do_flush_list_batch()

```
1729  static ulint buf_do_flush_list_batch(buf_pool_t *buf_pool, ulint min_n,
1730                                       lsn_t lsn_limit) {
1731    ulint count = 0;
1732    ulint scanned = 0;
1733
1734    /* Start from the end of the list looking for a suitable
1735    block to be flushed. */
1736    buf_flush_list_mutex_enter(buf_pool);
1737    ulint len = UT_LIST_GET_LEN(buf_pool->flush_list);
1738
1739    /* In order not to degenerate this scan to O(n*n) we attempt
1740    to preserve pointer of previous block in the flush list. To do
1741    so we declare it a hazard pointer. Any thread working on the
1742    flush list must check the hazard pointer and if it is removing
1743    the same block then it must reset it. */
1744    for (buf_page_t *bpage = UT_LIST_GET_LAST(buf_pool->flush_list);
1745         count < min_n && bpage != NULL && len > 0 &&
1746         bpage->oldest_modification < lsn_limit;
1747         bpage = buf_pool->flush_hp.get(), ++scanned) {
```

Get the last page from flush list

Flush List Flushing

- buf/buf0flu.cc: buf_do_flush_list_batch()

```
1748        buf_page_t *prev;
1749
1750        ut_a(bpage->oldest_modification > 0);
1751        ut_ad(bpage->in_flush_list);
1752
1753        prev = UT_LIST_GET_PREV(list, bpage);
1754        buf_pool->flush_hp.set(prev);
1755
1756  #ifdef UNIV_DEBUG
1757        bool flushed =
1758  #endif /* UNIV_DEBUG */
1759            buf_flush_page_and_try_neighbors(bpage, BUF_FLUSH_LIST, min_n, &count);
1760
1761        ut_ad(flushed || buf_pool->flush_hp.is_hp(prev));
1762
1763        --len;
1764      }
```

For all flushable pages within the flush area, flush them asynchronously

Flush List Flushing: Complete I/O

- After all the work for flushing is complete, the following function is called last to complete I/O
- buf/buf0buf.cc: buf_page_io_complete()

```
4657  bool buf_page_io_complete(buf_page_t *bpage, bool evict) {
                                        ...
4867      case BUF_IO_WRITE:
4868        /* Write means a flush operation: call the completion
4869        routine in the flush system */
4870
4871        buf_flush_write_complete(bpage);
```

Flush List Flushing: Complete I/O

- buf/buf0buf.cc: buf_page_io_complete()

```
4879        /* We decide whether or not to evict the page from the
4880        LRU list based on the flush_type.
4881        * BUF_FLUSH_LIST: don't evict
4882        * BUF_FLUSH_LRU: always evict
4883        * BUF_FLUSH_SINGLE_PAGE: evicti
4884        by the caller explicitly. */
4885        if (buf_page_get_flush_type(bpa
4886          evict = true;
4887          ut_ad(have_LRU_mutex);
4888        }
4889
4890        if (evict && buf_LRU_free_page(bpage, true)) {
4891          have_LRU_mutex = false;
4892        } else {
4893          mutex_exit(buf_page_get_mutex(bpage));
4894        }
4895        if (have_LRU_mutex) {
4896          mutex_exit(&buf_pool->LRU_list_mutex);
4897        }
```

Do not free the flushed page!
Keep it in LRU list as a clean page.

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: pc_flush_slot()

```
2699  finish:
2700    mutex_enter(&page_cleaner->mutex);
2701  finish_mutex:
2702    page_cleaner->n_slots_flushing--;
2703    page_cleaner->n_slots_finished++;
2704    slot->state = PAGE_CLEANER_STATE_FINISHED;
2705
2706    slot->flush_lru_time += lru_tm;
2707    slot->flush_list_time += list_tm;
2708    slot->flush_lru_pass += lru_pass;
2709    slot->flush_list_pass += list_pass;
2710
2711    if (page_cleaner->n_slots_requested == 0 &&
2712        page_cleaner->n_slots_flushing == 0) {
2713      os_event_set(page_cleaner->is_finished);
2714    }
2715  }
```

Finishing flushing

```
2717    ulint ret = page_cleaner->n_slots_requested;
2718
2719    mutex_exit(&page_cleaner->mutex);
2720
2721    return (ret);
2722  }
```

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: buf_flush_page_coordinator_thread()

```
3074        /* only coordinator is using these counters,
3075        so no need to protect by lock. */
3076        page_cleaner->flush_time += ut_time_ms() - tm;
3077        page_cleaner->flush_pass++;
3078
3079        /* Wait for all slots to be finished */
3080        ulint n_flushed_lru = 0;
3081        ulint n_flushed_list = 0;
3082
3083        pc_wait_finished(&n_flushed_lru, &n_flushed_list);
3084
3085        if (n_flushed_list > 0 || n_flushed_lru > 0) {
3086          buf_flush_stats(n_flushed_list, n_flushed_lru);
3087        }
```

Wait until all flush requests are finished

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: pc_wait_finished()

```
2730  static bool pc_wait_finished(ulint *n_flushed_lru, ulint *n_flushed_list) {
2731    bool all_succeeded = true;
2732
2733    *n_flushed_lru = 0;
2734    *n_flushed_list = 0;
2735
2736    os_event_wait(page_cleaner->is_finished);
2737
2738    mutex_enter(&page_cleaner->mutex);
2739
2740    ut_ad(page_cleaner->n_slots_requested == 0);
2741    ut_ad(page_cleaner->n_slots_flushing == 0);
2742    ut_ad(page_cleaner->n_slots_finished == page_cleaner->n_slots);
```

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: pc_wait_finished()

For all page cleaner slots

```
2744  for (ulint i = 0; i < page_cleaner->n_slots; i++) {
2745    page_cleaner_slot_t *slot = &page_cleaner->slots[i];
2746
2747    ut_ad(slot->state == PAGE_CLEANER_STATE_FINISHED);
2748
2749    *n_flushed_lru += slot->n_flushed_lru;
2750    *n_flushed_list += slot->n_flushed_list;
2751    all_succeeded &= slot->succeeded_list;
2752
2753    slot->state = PAGE_CLEANER_STATE_NONE;
2754
2755    slot->n_pages_requested = 0;
2756  }
2757
2758  page_cleaner->n_slots_finished = 0;
2759
2760  os_event_reset(page_cleaner->is_finished);
2761
2762  mutex_exit(&page_cleaner->mutex);
```

Aggregate the number of total flushed pages

Reset the state of slot

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: buf_flush_page_coordinator_thread()

```
3093        n_evicted += n_flushed_lru;
3094        n_flushed_last += n_flushed_list;
3095
3096        n_flushed = n_flushed_lru + n_flushed_list;
3097
3098        if (n_flushed_lru) {
3099          MONITOR_INC_VALUE_CUMULATIVE(
3100              MONITOR_LRU_BATCH_FLUSH_TOTAL_PAGE, MONITOR_LRU_BATCH_FLUSH_COUNT,
3101              MONITOR_LRU_BATCH_FLUSH_PAGES, n_flushed_lru);
3102        }
3103
3104        if (n_flushed_list) {
3105          MONITOR_INC_VALUE_CUMULATIVE(
3106              MONITOR_FLUSH_ADAPTIVE_TOTAL_PAGE, MONITOR_FLUSH_ADAPTIVE_COUNT,
3107              MONITOR_FLUSH_ADAPTIVE_PAGES, n_flushed_list);
3108        }
```

Set the number of total flushed pages (LRU + flush list flushing)

Page Cleaner: Coordinator Thread

- buf/buf0flu.cc: buf_flush_page_coordinator_thread()

```
3110        } else if (ret_sleep == OS_SYNC_TIME_EXCEEDED) {
3111          /* no activity, slept enough */
3112          buf_flush_lists(PCT_IO(100), LSN_MAX, &n_flushed);
3113
3114          n_flushed_last += n_flushed;
3115
3116          if (n_flushed) {
3117            MONITOR_INC_VALUE_CUMULATIVE(MONITOR_FLUSH_BACKGROUND_TOTAL_PAGE,
3118                                         MONITOR_FLUSH_BACKGROUND_COUNT,
3119                                         MONITOR_FLUSH_BACKGROUND_PAGES, n_flushed);
3120          }
3121
3122        } else {
3123          /* no activity, but woken up by event */
3124          n_flushed = 0;
3125        }
```

3rd case: Nothing has been changed, but OS_SYNC_TIME_EXCEEDED is set

4th case: No activity

MYSQL CHECKPOINTS

Types of checkpoints

- **Sharp** checkpoint (at shutdown)
  - Flushing all modified pages for *committed* transactions to disk
  - Writing down the LSN of the most recent committed transaction
  - All flushed pages is consistent as of a single point in time (the checkpoint LSN) ➔ "sharp"

- **Fuzzy** checkpoint (at normal time)
  - Flushing pages as time passes (flush list flushing)
  - Flushed pages might not all be consistent with each other as of a single point in time ➔ "fuzzy"

Types of checkpoints

- **Periodical** checkpoint (every X seconds)
  - *Before 8.0*, the master thread was doing periodical checkpoints (every 7s)
  - *Since 8.0*, the log checkpointer thread is responsible for periodical checkpoints (every innodb_log_checkpoint_every ms)

Log Checkpointer Thread

- Checking if a checkpoint write is required
  - To decrease checkpoint age before it gets too big
- Checking if synchronous flush of dirty pages should be forced on page cleaner threads, because of space in redo log or age of the oldest page
- Writing checkpoints
  - It’s the only thread allowed to do it!

Log Checkpointer Thread Init

- log/log0log.cc: log_start_background_threads()

```
704  void log_start_background_threads(log_t &log) {
                                    ...
715    log.closer_thread_alive = true;
716    log.checkpointer_thread_alive = true;
717    log.writer_thread_alive = true;
718    log.flusher_thread_alive = true;
719    log.write_notifier_thread_alive = true;
720    log.flush_notifier_thread_alive = true;
                                    ...
726    os_thread_create(log_checkpointer_thread_key, log_checkpointer, &log);
```

Create the log checkpointer thread

Log Checkpointer Thread

- log/log0chkp.cc: log_checkpointer()

```
837  void log_checkpointer(log_t *log_ptr) {
                                    ...
845    while (true) {
846      auto do_some_work = [&log] {
847
848        ut_ad(log_checkpointer_mutex_own(log));
849
850        /* We will base our next decisions on maximum lsn
851        available for creating a new checkpoint. It would
852        be great to have it updated beforehand. Also, this
853        is the only thread that relies on that value, so we
854        don't need to update it in other threads. */
855        log_update_available_for_checkpoint_lsn(log);
856
857        /* Consider flushing some dirty pages. */
858        const bool sync_flushed = log_consider_sync_flush(log);
```

Check if it has some work to do

Log Checkpointer Thread

- log/log0chkp.cc: log_checkpointer()

```
862         /* Consider writing checkpoint. */
863         const bool checkpointed = log_consider_checkpoint(log);
864
865         if (sync_flushed || checkpointed) {
866           return (true);
867         }
868
869         return (false);
870       };
871
872       const auto sig_count = os_event_reset(log.checkpointer_event);
873
874       if (!do_some_work()) {
875         log_checkpointer_mutex_exit(log);
876
877         os_event_wait_time_low(log.checkpointer_event, 10 * 1000, sig_count);
878
879         log_checkpointer_mutex_enter(log);
880
881       } else {
```

Log Checkpointer Thread

- log/log0chkp.cc: log_consider_checkpoint()

```
804  static bool log_consider_checkpoint(log_t &log) {
805    ut_ad(log_checkpointer_mutex_own(log));
806
807    if (!log_should_checkpoint(log)) {
808      return (false);
809    }
810
811    /* It's clear that a new checkpoint should be written.
812    So do write back the dynamic metadata. Since the checkpointer
813    mutex is low-level one, it has to be released first. */
814    log_checkpointer_mutex_exit(log);
                              ...
833    log_checkpoint(log);
834    return (true);
835  }
```

Checks if checkpoint should be written

Log Checkpointer Thread

- log/log0chkp.cc: log_should_checkpoint()

```
750  static bool log_should_checkpoint(log_t &log) {
                                  ...
787    /* Update checkpoint_lsn stored in header of log files if:
788            a) more than 1s elapsed since last checkpoint
789            b) checkpoint age is greater than max_checkpoint_age_async
790            c) it was requested to have greater checkpoint_lsn,
791               and oldest_lsn allows to satisfy the request */
792
793    if ((log.periodical_checkpoints_enabled &&
794         checkpoint_time_elapsed >= srv_log_checkpoint_every * 1000ULL) ||
795        checkpoint_age >= log.max_checkpoint_age_async ||
796        (requested_checkpoint_lsn > last_checkpoint_lsn &&
797         requested_checkpoint_lsn <= oldest_lsn)) {
798      return (true);
799    }
800
801    return (false);
802  }
```

Log Checkpointer Thread

- log/log0chkp.cc: log_consider_checkpoint()

```
804  static bool log_consider_checkpoint(log_t &log) {
805    ut_ad(log_checkpointer_mutex_own(log));
806
807    if (!log_should_checkpoint(log)) {
808      return (false);
809    }
810
811    /* It's clear that a new checkpoint should be written.
812    So do write back the dynamic metadata. Since the checkpointer
813    mutex is low-level one, it has to be released first. */
814    log_checkpointer_mutex_exit(log);
```

...

```
833    log_checkpoint(log);
834    return (true);
835  }
```

Makes a checkpoint;
Note that this function **does not flush dirty blocks from the buffer pool**. It only checks what is LSN of the oldest modification in the buffer pool, and **writes information about the LSN in log files**.

Log Checkpointer Thread

- log/log0chkp.cc: log_checkpointer()

```
862         /* Consider writing checkpoint. */
863         const bool checkpointed = log_consider_checkpoint(log);
864
865         if (sync_flushed || checkpointed) {
866           return (true);
867         }
868
869         return (false);
870       };
871
872       const auto sig_count = os_event_reset(log.checkpointer_event);
873
874       if (!do_some_work()) {
875         log_checkpointer_mutex_exit(log);
876
877         os_event_wait_time_low(log.checkpointer_event, 10 * 1000, sig_count);
878
879         log_c
880
881       } else {
```

If it has no work to do

Waits for the checkpoint event until it is in the signaled state or a timeout (10ms) is exceeded

Log Checkpointer Thread

- log/log0chkp.cc: log_checkpointer()

```
881       } else {
882         log_checkpointer_mutex_exit(log);
883
884         os_thread_sleep(0);
885
886         log_checkpointer_mutex_enter(log);
887       }
888
889       /* Check if we should close the thread. */
890       if (log.should_stop_threads.load() && !log.closer_thread_alive.load()) {
891         break;
892       }
893     }
```

Log Checkpointer Thread

- log/log0chkp.cc: log_checkpointer()

```
881       } else {
882         log_checkpointer_mutex_exit(log);
883
884         os_thread_sleep(0);
885
886         log_checkpointer_mutex_enter(log);
887       }
888
889       /* Check if we should close the thread. */
890       if (log.should_stop_threads.load() && !log.closer_thread_alive.load()) {
891         break;
892       }
893     }
```

Reference

[1] “MySQL 5.7 Reference Manual”, MySQL, https://dev.mysql.com/doc/refman/5.7/en/

[2] Jeremy Cole, “InnoDB”, https://blog.jcole.us/innodb/

[3] Laurynas Biveinis, Alexey Stroganov, “Percona Server for MySQL 5.7: Key Performance Algorithms”, Percona, https://www.percona.com/resources/webinars/percona-server-mysql-57-key-performance-algorithms