



Максим Милютин

milyutinma@gmail.com

Разделяемый буфер

Внутрянка. Часть 1

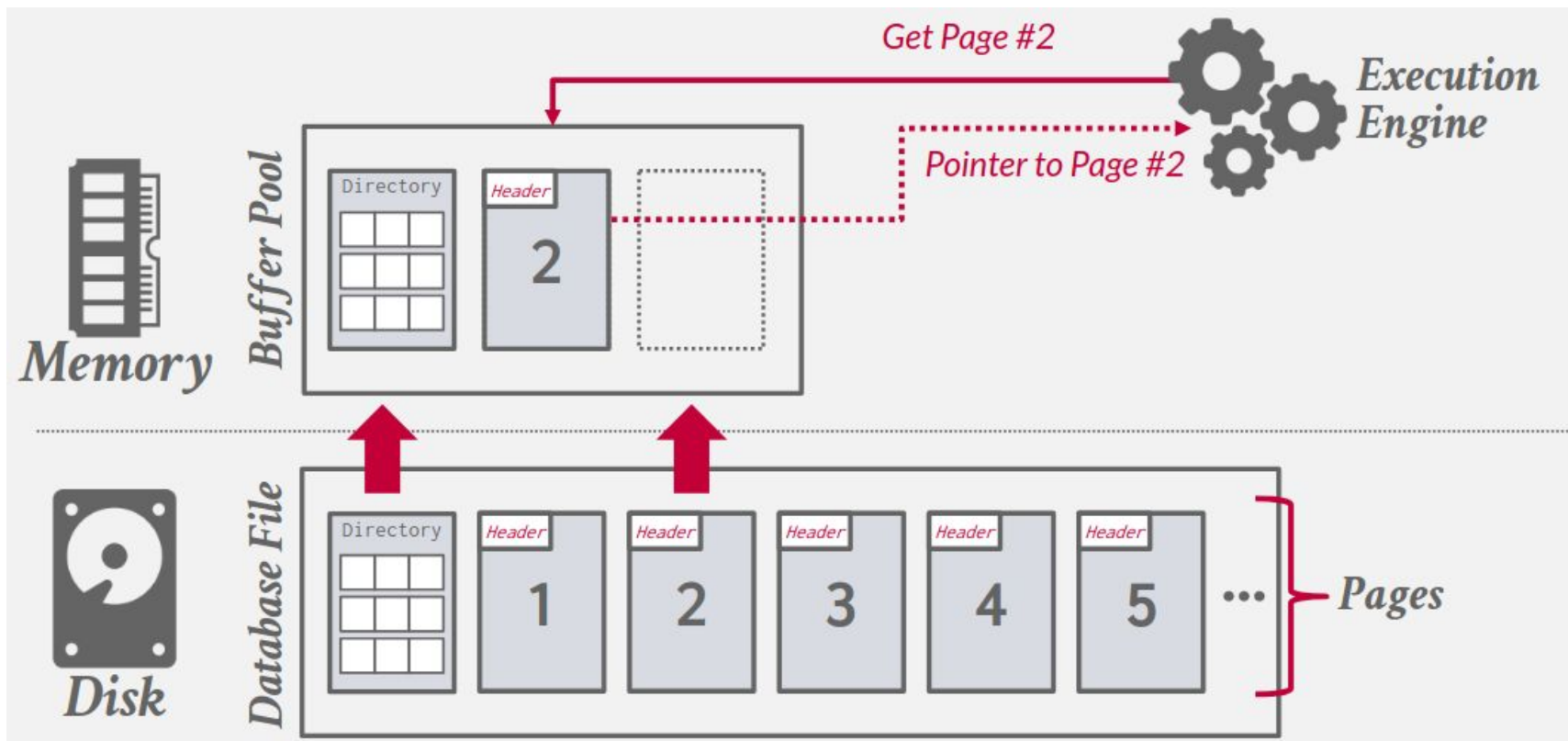
О докладчике

- R&D SQL движков
 - улучшение планировщика ML методами
 - работа с распределенными базами
- Более 8 лет работы с базами данных и PostgreSQL
 - SQL прикладная разработка в e-comm
 - DBA и консалтинг решений вокруг PostgreSQL
 - Разработка патчей к PostgreSQL и Greenplum и расширений к ним
 - R&D разработка вокруг OpenGauss
- Веду курс по разработке СУБД для студентов НГУ и открыто
 - ссылка для вступления в telegram группу <https://t.me/+eIO3me6xzFUyNjFi>

Агенда

- От простой модели к актуальному состоянию
- Нюансы реализации
- IO операции
- Оптимизирующие конструкции
- Стратегия вытеснения *ring buffer* (vs. *clock sweep*)
- Мониторинг и инструментация
- *Scan sharing*

Буферный кэш



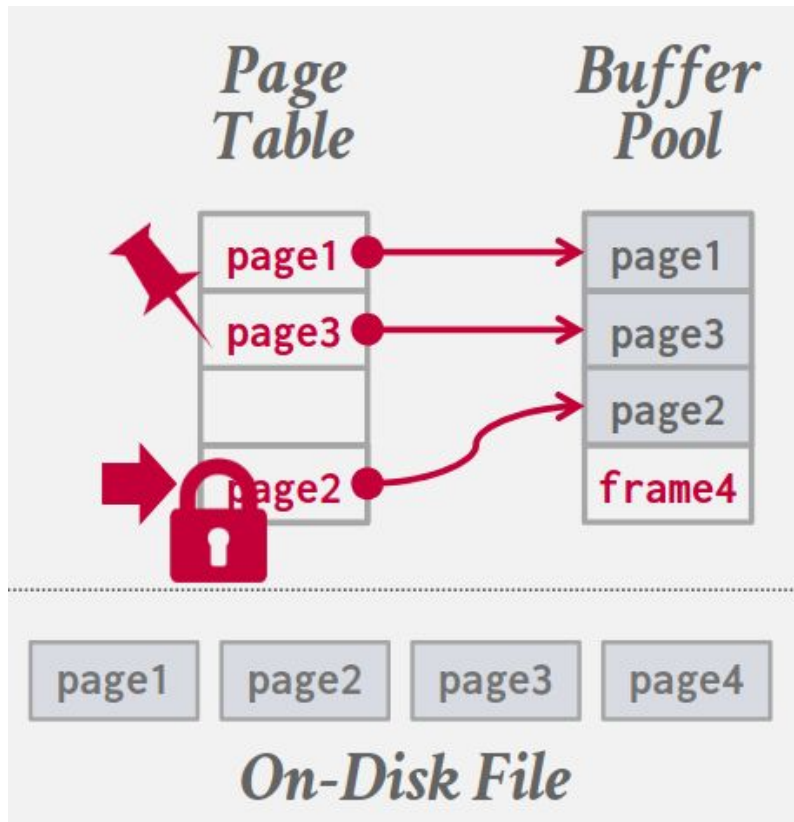
mapping: *pageId* → *pointer to page* in **virtual address space**

Буферный кэш. PageID

```
/*
 * Buffer tag identifies which disk block the buffer contains.
 * ...
 */
typedef struct buftag
{
    Oid          spcOid;          /* tablespace oid */
    Oid          dbOid;          /* database oid */
    RelFileNumber relNumber;     /* relation file number */
    ForkNumber   forkNum;        /* fork number */
    BlockNumber  blockNum;       /* blknum relative to begin of reln */
} BufferTag;
```

src/include/storage/buf_internals.h

Структура буферного кэша



Page table - разделяемая хэш-таблица (*pageId* → *address*) для ведения страниц в кэше. Хранит:

- dirty флаг
- pin и reference счётчики
- состояние ячейки
- информацию о доступе

Защищена мьютексом (LWLock`ом)

Ненулевой *pin*-счётчик запрещает вытеснение страницы (“страница *заpinена*”)

Локи (hwlocks) vs. Латчи (lwlocks, мьютексы)

	<i>Locks</i>	<i>Latches</i>
Separate ...	User transactions	Threads
Protect ...	Database contents	In-memory data structures
During ...	Entire transactions	Critical sections
Modes ...	Shared, exclusive, update, intention, escrow, schema, etc.	Read, writes, (perhaps) update
Deadlock ...	Detection & resolution	Avoidance
... by ...	Analysis of the waits-for graph, timeout, transaction abort, partial rollback, lock de-escalation	Coding discipline, “lock leveling”
Kept in ...	Lock manager’s hash table	Protected data structure

Буферный кэш. Три базовые операции

- **Buffer access**

доступ к закэшированной странице

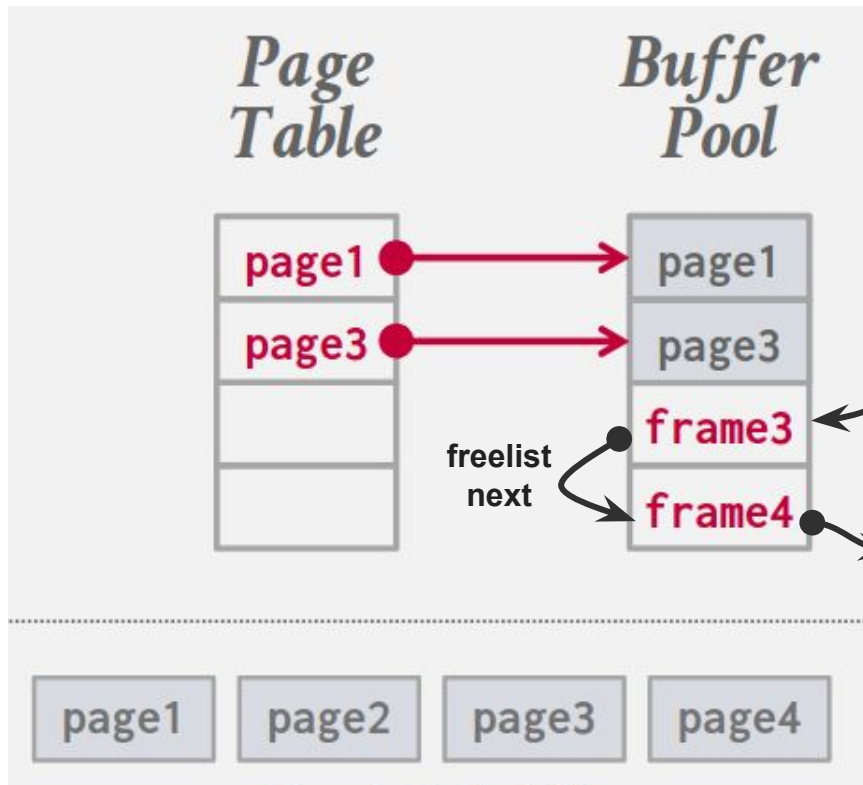
- **Buffer allocation**

выделение буфера из свободной ячейки

- **Buffer replacement**

выделение буфера с вытеснением существующего

Аллокация кэша



Freelist - список свободных ячеек буфера

Пополняется при удалении базы, схемы, таблицы, васиум

Защищен **спинлоком**

```
/* Acquire the spinlock to remove element from the freelist */
SpinLockAcquire(&StrategyControl->buffer_strategy_lock);

if (StrategyControl->firstFreeBuffer < 0)
{
    SpinLockRelease(&StrategyControl->buffer_strategy_lock);
    break;
}

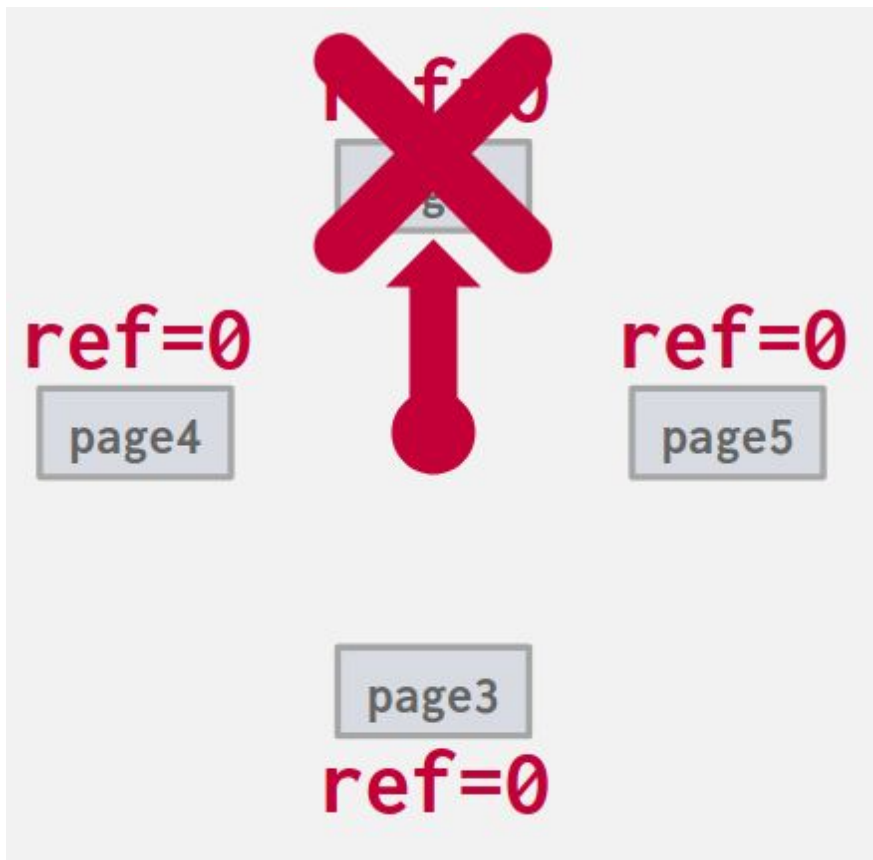
buf = GetBufferDescriptor(StrategyControl->firstFreeBuffer);
Assert(buf->freeNext != FREENEXT_NOT_IN_LIST);

/* Unconditionally remove buffer from freelist */
StrategyControl->firstFreeBuffer = buf->freeNext;
buf->freeNext = FREENEXT_NOT_IN_LIST;

/*
 * Release the lock so someone else can access the freelist while
 * we check out this buffer.
 */
SpinLockRelease(&StrategyControl->buffer_strategy_lock);
```

src/backend/storage/buffer/freelist.c

Clock. Алгоритм вытеснения страницы



Разделяемый **reference** счётчик
внутри элемента *page table*

Атомарный инкремент

```
static inline uint32
ClockSweepTick(void)
{
    ...
    victim =
    pg_atomic_fetch_add_u32(&nextVictimBuffer, 1);

    if (victim >= NBuffers)
    {
        ...
        victim = victim % NBuffers;
        ...
    }
    return victim;
}
```

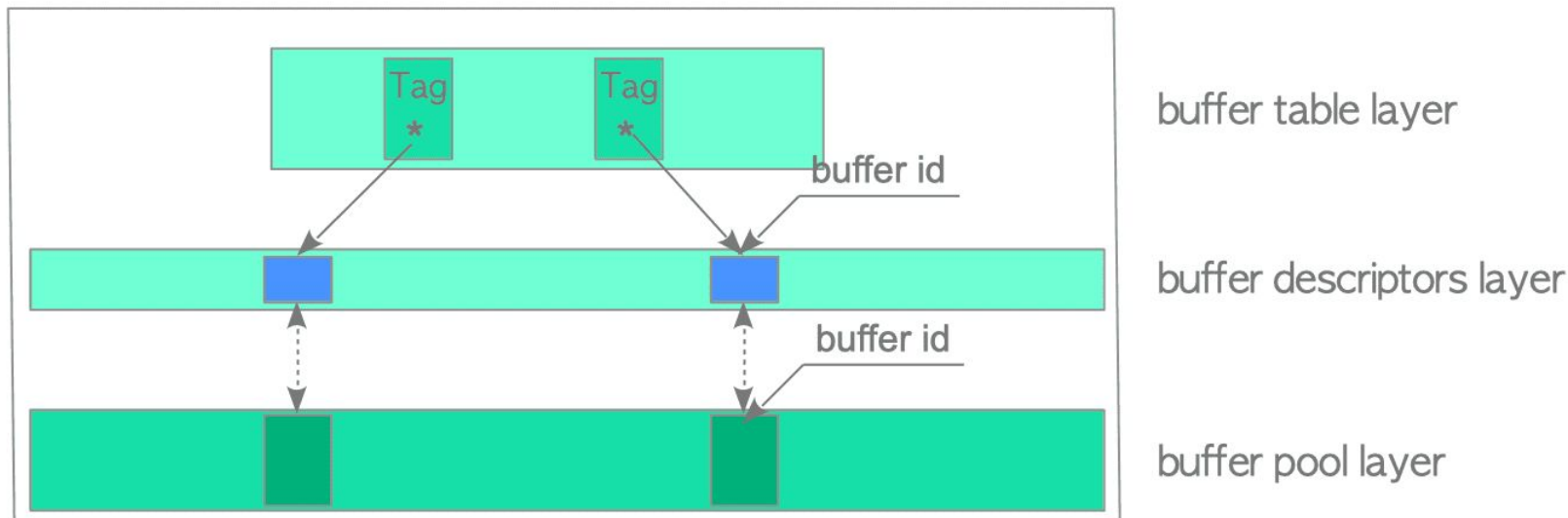
src/backend/storage/buffer/freelist.c

Clock. Алгоритм вытеснения страницы

- Метаданные (*pin* и *reference* счётчики) ячеек хранятся в элементах хэш-таблицы ***PageTable***
- При обходе буфера требуется **доступ** к метаданным **по номеру ячейки**
 - обратный поиск по элементам хэш-таблицы
- Необходим третий промежуточный слой для хранения метаданных с индексацией по номеру ячейки

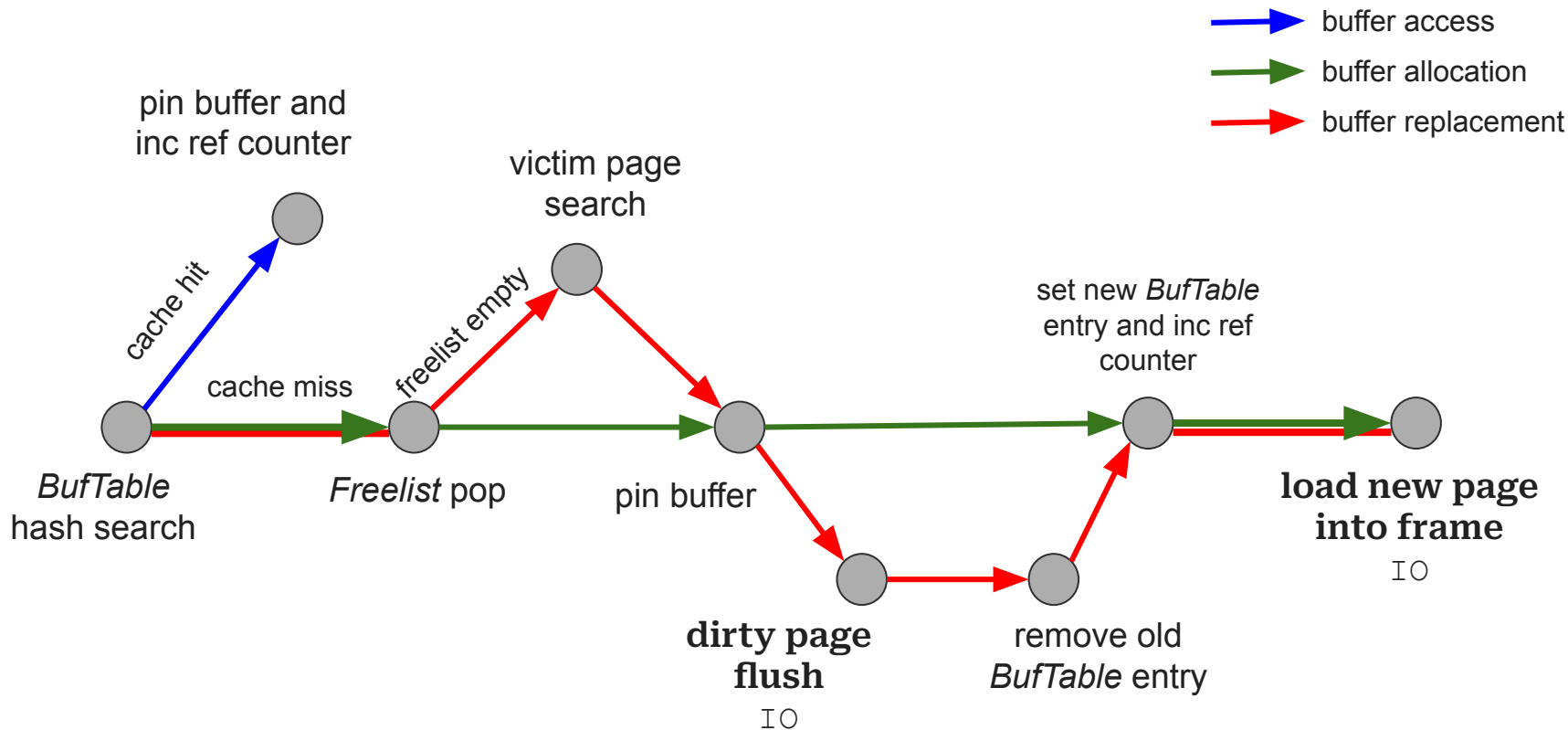
Обновлённая структура буферного кэша

Обходя буфер, метаданные ячейки следует искать в промежуточном слое
Buffer Descriptors layer



Каждая ячейка буфер-дескриптора защищена **СВОИМ СПИНЛОКОМ**

Буферный кэш. Workflow



- **Buffer access**

- поиск страницы в buffer table под мьютексом на чтение
- пин и инкремент reference счётчика в буфер-дескрипторе под спинлоком дескриптора

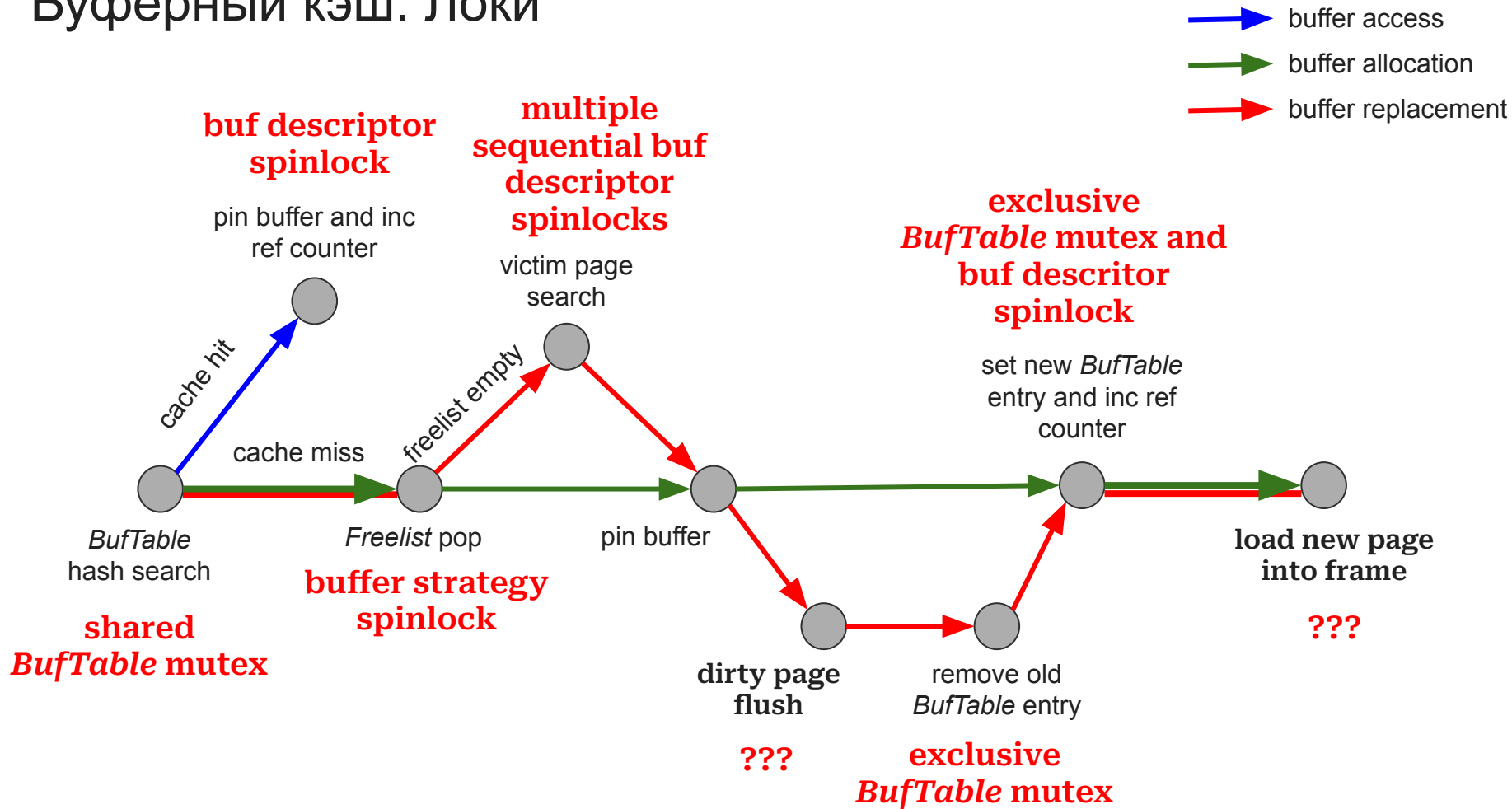
- **Buffer allocation**

- поиск страницы в buffer table под мьютексом на чтение
- изъятие свободной ячейки из freelist под freelist-спинлоком
- пин буфер-дескриптора под спинлоком дескриптора
- установка соответствия *BufferTag* → *BufferNum* (pageid → pointer to page) в buffer table под мьютексом на запись
- *загрузка новой страницы в новую ячейку буфера*

- **Buffer replacement**

- поиск страницы в buffer table под мьютексом на чтение
- последовательный обход буфер-дескрипторов на поиск ячейки-“жертвы” с захватом спинлоков соответствующих дескрипторов
- пин буфер-дескриптора под спинлоком дескриптора
- *вытеснение грязной страницы*
- удаление старого *BufferTag* и установка нового соответствия *BufferTag* → *BufferNum* в buffer table под мьютексом на запись
- *загрузка новой страницы в выбранную ячейку буфера*

Буферный кэш. Локи

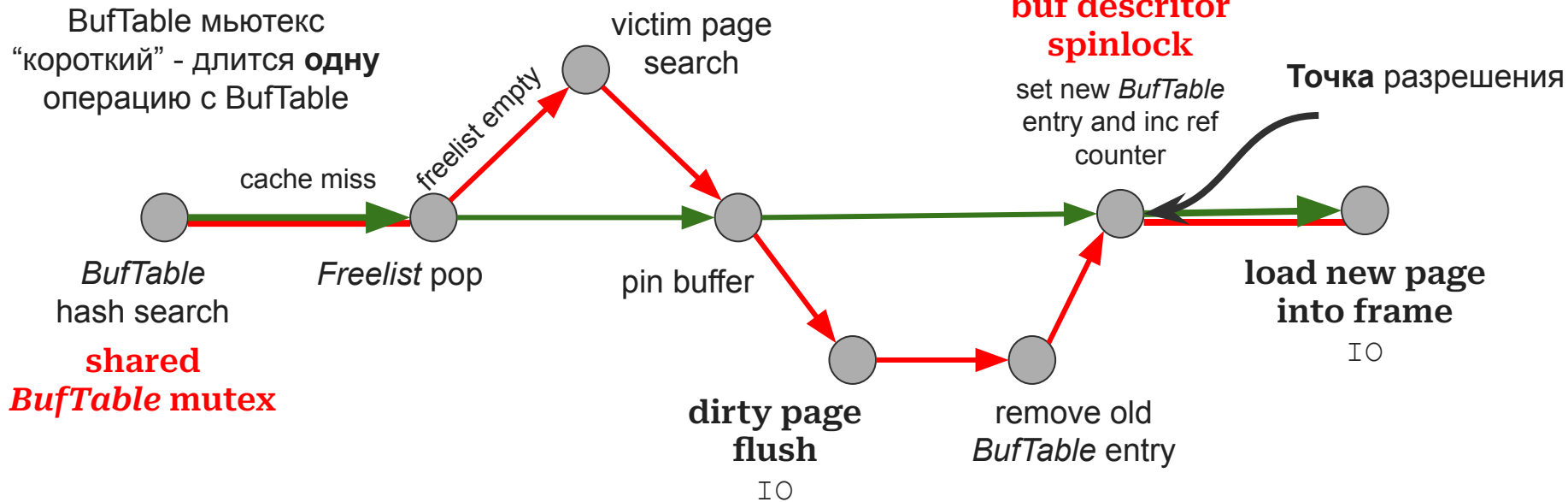


Буферный кэш. Конкуренция при allocation/replacement

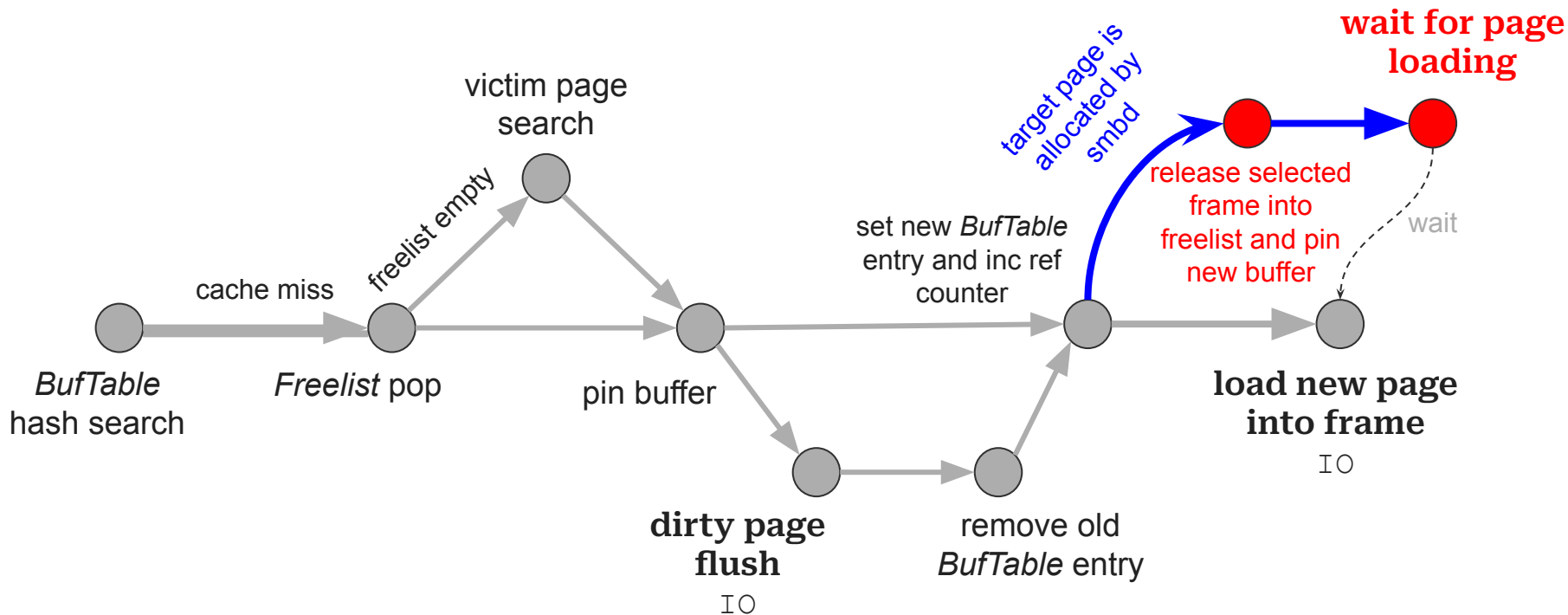
Воркеры, аллоцирующие одну и ту же страницу



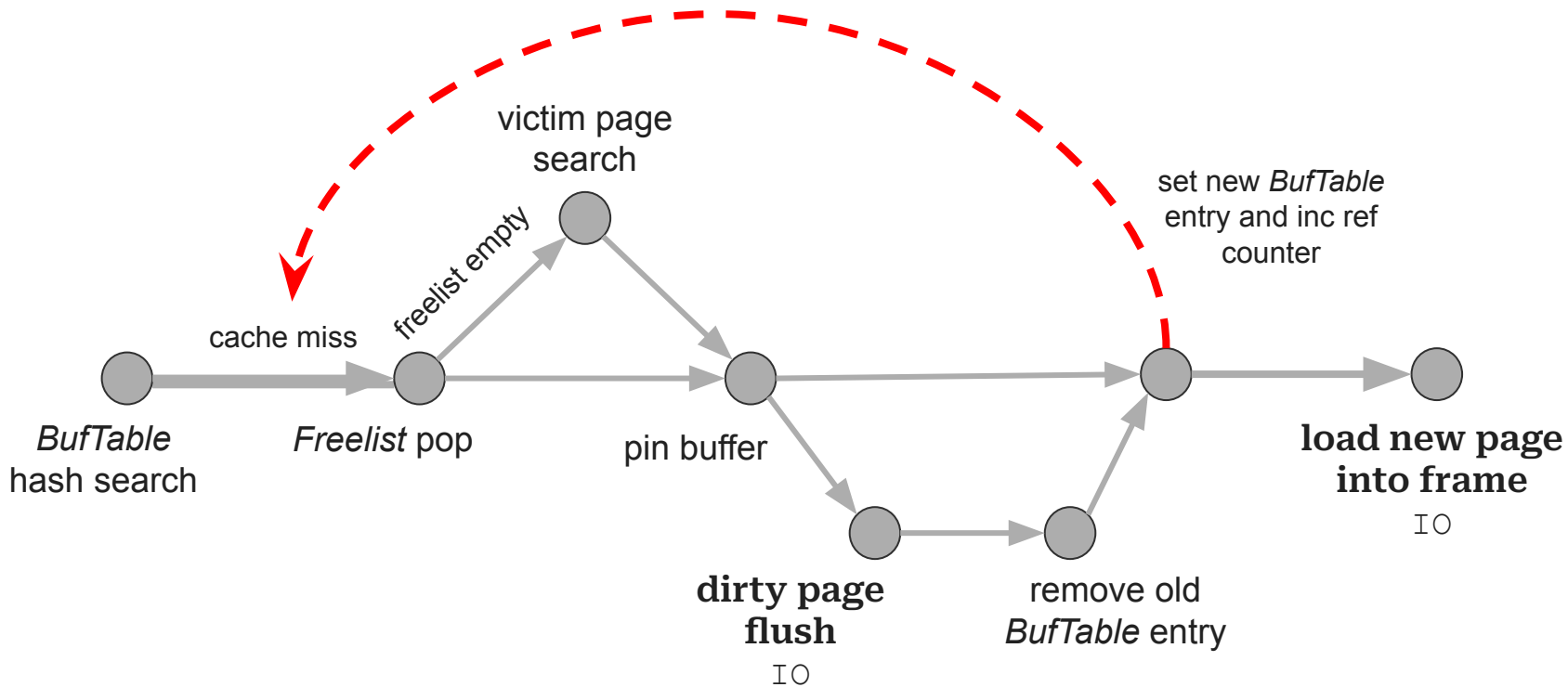
→ buffer allocation
→ buffer replacement



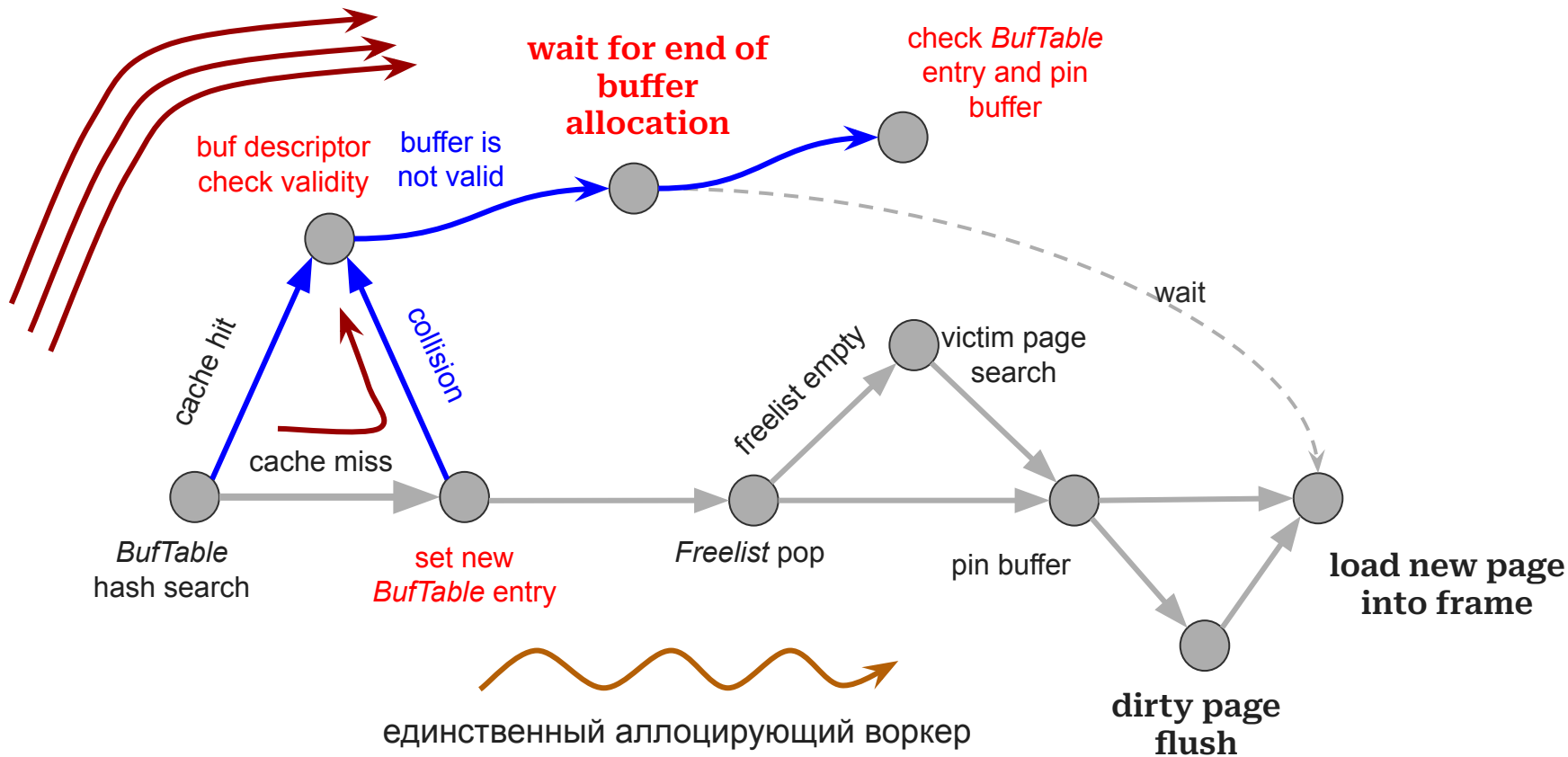
Буферный кэш. Конкуренция при allocation/replacement



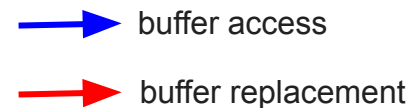
Буферный кэш. Избежание конкуренции при allocation/replacement



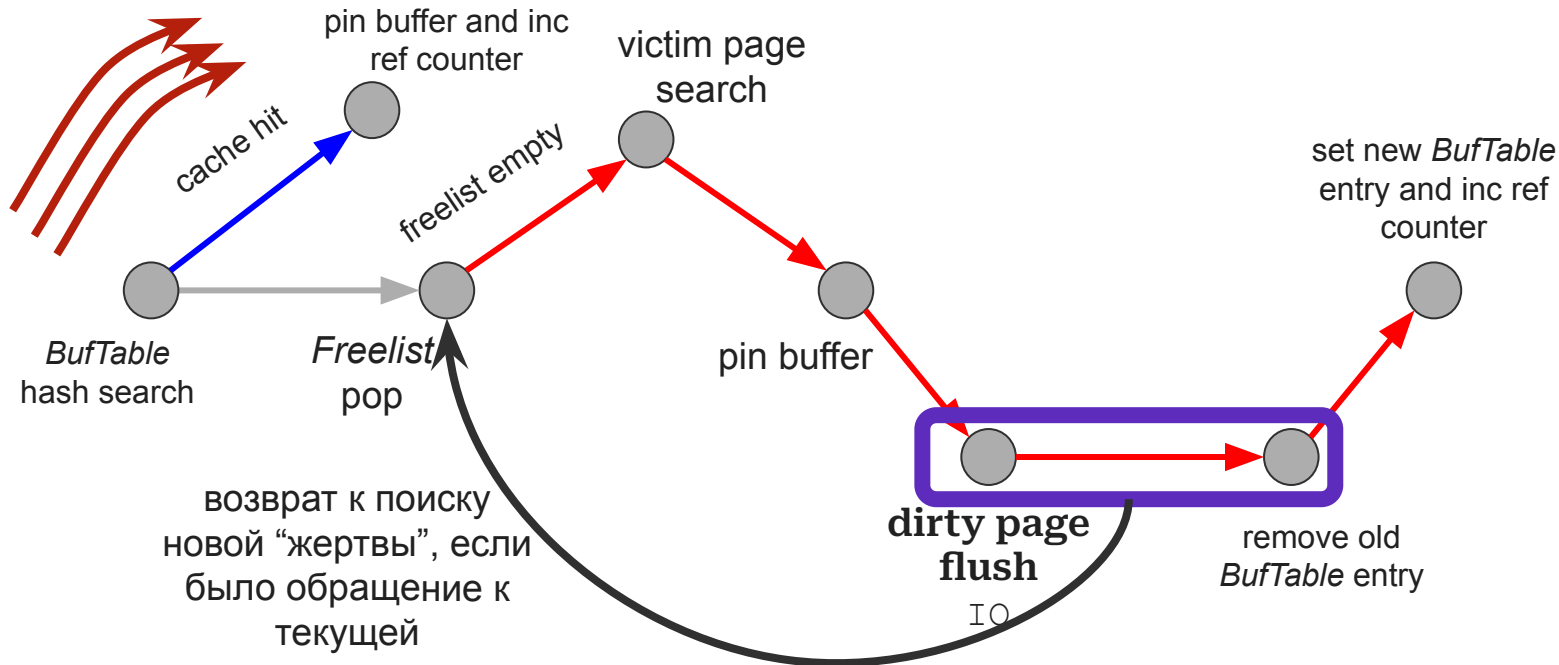
Буферный кэш. Избежание конкуренции при allocation/replacement



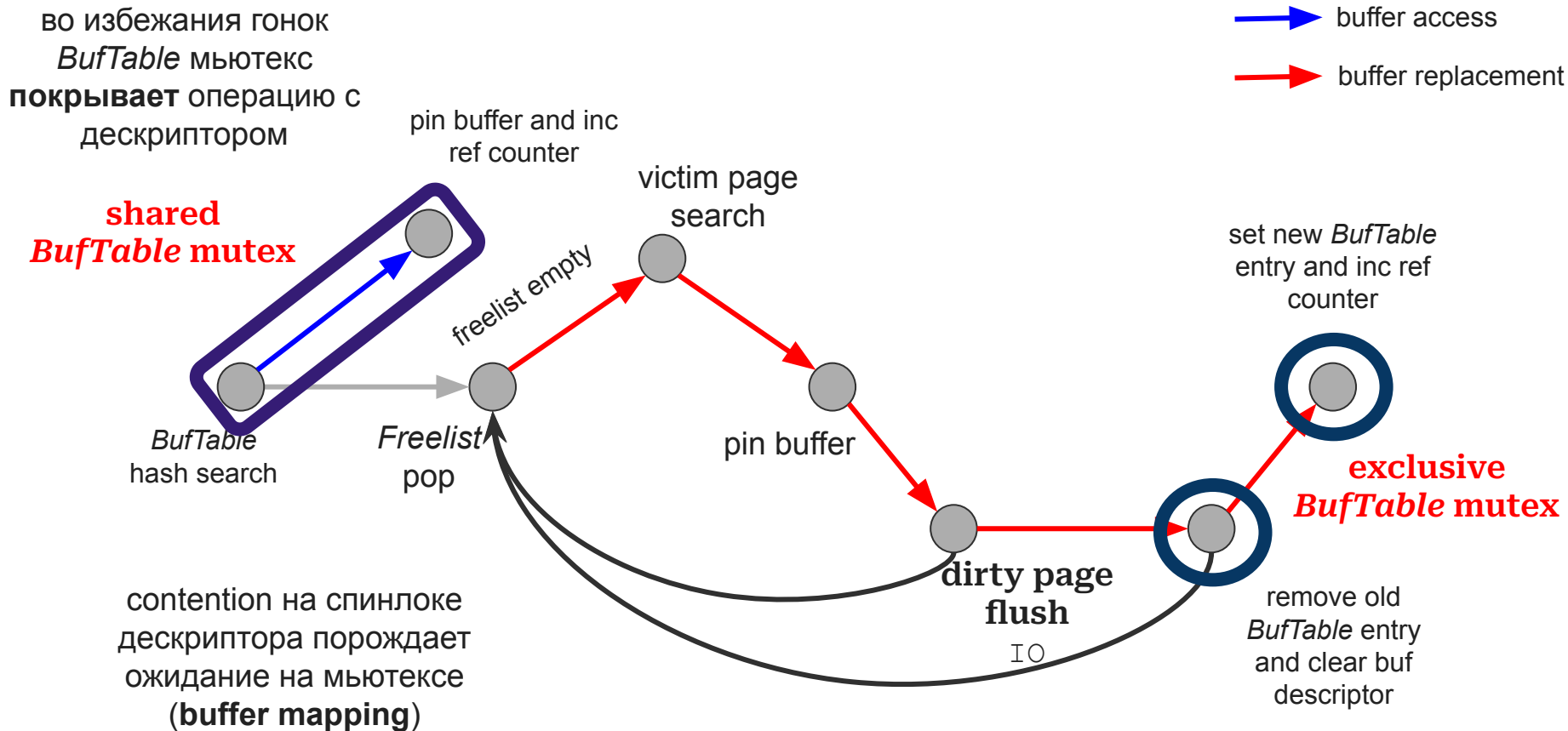
Буферный кэш. Одновременная работа с буфером при replacement



воркеры, обращающиеся
к буферу-"жертве"



Буферный кэш. Покрывтие *BufTable* мьютекса



Буферный кэш. Покрытие *BufTable* мьютекса

Buffer access

BufTable hash search

Pin buffer

Increment ref counter



Buffer invalidation

Check on buffer pinned by smbd

Clear buf descriptor

Remove *BufTable* hash entry

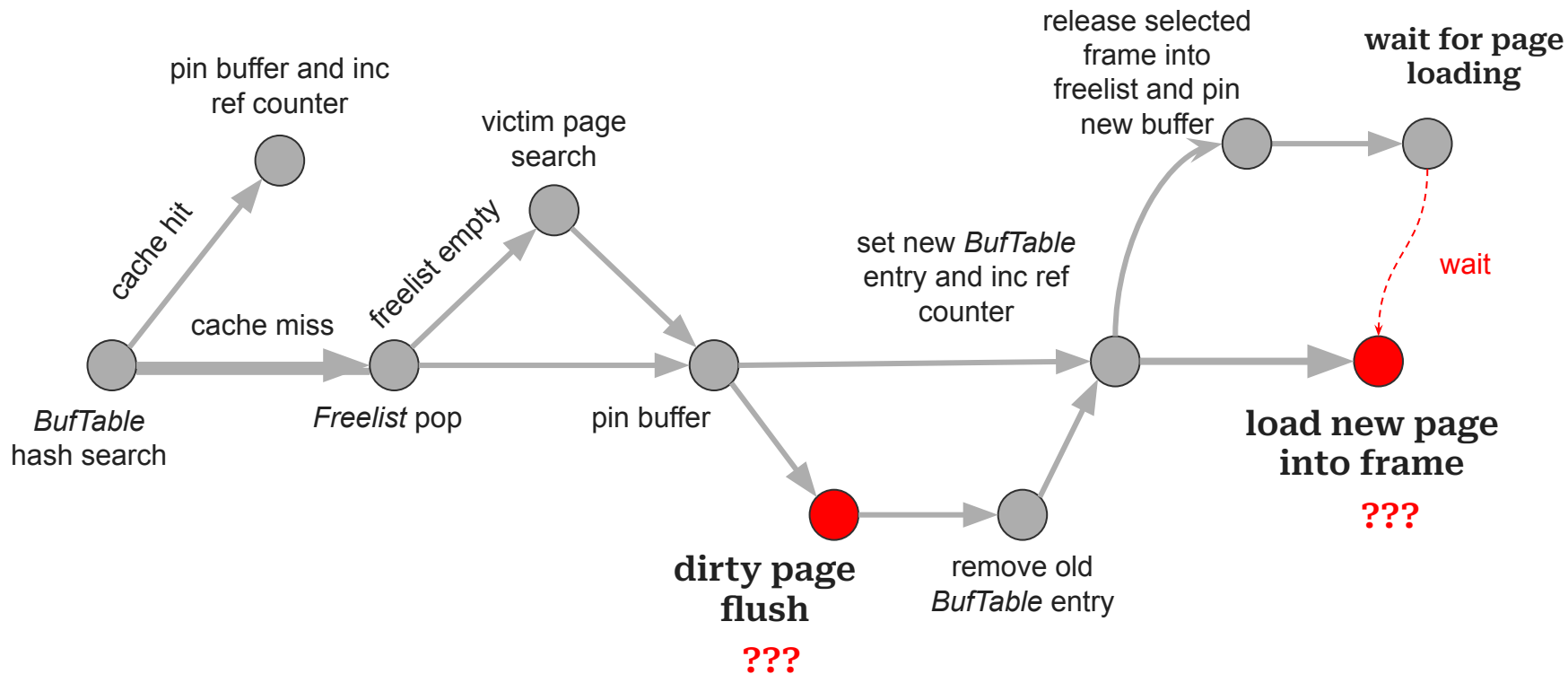


Наличие гонок при разделении взятия
мьютекса на *BufTable* и спинлока на текущий
дескриптор

Буферный кэш. Свободные буфера вне *freelist*

- Свободные буфера необязательно держать во *freelist*
 - их **значительно более долгий** поиск будет через Clock-алгоритм
- При Clock-алгоритме признак “свободности” не проверяется
 - “свободная жертва”, находясь во *freelist*, оттуда не извлекается
- Во *freelist* могут находиться уже выделенные буфера
 - если будущая “жертва” была добавлена во *freelist* сторонним процессом во время Clock-обхода
- При изъятии из *freelist* необходимо проверять *pin* и *reference* счётчики
 - под соответствующим спинлоком

Буферный кэш. IO операции



Вытеснение грязной страницы

```
/*
 * If the buffer was dirty, try to write it out
 */
if (buf_state & BM_DIRTY)
{
    ...
    /*
     * We need a share-lock on the buffer contents to write it out (else
     * we might write invalid data, eg because someone else is compacting
     * the page contents while we write)
     */
    content_lock = BufferDescriptorGetContentLock(buf_hdr);
    if (!LWLockConditionalAcquire(content_lock, LW_SHARED))
    {
        /*
         * Someone else has locked the buffer, so give it up and loop back
         * to get another one.
         */
        UnpinBuffer(buf_hdr);
        goto again;
    }
    ...
    /* OK, do the I/O */
    FlushBuffer(buf_hdr, NULL, IOOBJECT_RELATION, io_context);
    LWLockRelease(content_lock);

    ScheduleBufferTagForWriteback(&BackendWritebackContext, io_context,
                                   &buf_hdr->tag);
}
```

src/backend/storage/buffer/bufmgr.c

запись под разделяемым
buffer_content мьютексом

возврат к выбору новой “жертвы”, если
мьютекс захвачен (кто-то уже
работает с буфером)

запись страницы **не сквозная**:
задача checkpointer`у на
физический сброс (*fsync*)
страницы

Сброс страницы

```
/*
 * FlushBuffer
 *     Physically write out a shared buffer.
 *
 * NOTE: this actually just passes the buffer contents to the kernel; the real write to disk won't happen until the
kernel feels like it. This is okay from our point of view since we can redo the changes from WAL. However, we will
need to force the changes to disk via fsync before we can checkpoint WAL.
 *
 * The caller must hold a pin on the buffer and have share-locked the buffer contents. (Note: a share-lock does not
prevent updates of hint bits in the buffer, so the page could change while the write is in progress, but we assume
that that will not invalidate the data written.)
 * ...
 */
static void
FlushBuffer(BufferDesc *buf, SMgrRelation reln, IOObject io_object,
            IOContext io_context)
{
    ...
    /*
     * Force XLOG flush up to buffer's LSN. This implements the basic WAL rule that log updates must hit disk before
any of the data-file changes they describe do.
     * ...
     */
    if (buf_state & BM_PERMANENT)
        XLogFlush(recptr);
    ...
    smgrwrite(reln,
              BufTagGetForkNum(&buf->tag),
              buf->tag.blockNum,
              bufToWrite,
              false);
    ...
}
```

В процессе записи может происходить
некритичная установка хинт битов
для таплов

Сброс WAL до позиции последней
модификации страницы: для
обеспечения **durability**

Буферный кэш. Вытеснение грязной страницы

- Влечёт максимум 2 IO операции (сброс WAL и запись страницы)
 - в Linux запись страницы в pagescache ОС
- Помимо записи сброс WAL может породить ожидания на:
 - **WALInsert** мьютексе при получении крайней позиции в WAL-буфере
 - **WALWrite** мьютексе при попадании на уже запущенный процесс сброса WAL
 - таймауте (параметр **commit_delay**) группового коммита
- Более хитрые стратегии вытеснения откладывают выбор грязных страниц
 - **write awareness** свойство buffer replacement алгоритмов
 - оставить сброс сторонним рабочим процессам **bgwriter**
 - имеются зачатки (функция **StrategyRejectBuffer()**) при массовом чтении из **ring buffer**

Буферный кэш. Загрузка страницы в буфер

Provide vectored variant of ReadBuffer().

author Thomas Munro <tmunro@postgresql.org>
Tue, 2 Apr 2024 11:03:08 +0000 (00:03 +1300)
committer Thomas Munro <tmunro@postgresql.org>
Tue, 2 Apr 2024 11:23:20 +0000 (00:23 +1300)
commit 210622c60e1a9db2e2730140b8106ab57d259d15
tree 9c8de4c53e6cd36fd48ac078d45037e5e8623e23 [tree](#)
parent 13b3b62746ec8bd9c8e3f0bc23862f1172996333 [commit](#) | [diff](#)

Provide `vectored variant` of `ReadBuffer()`.

Break `ReadBuffer()` up into two steps. `StartReadBuffers()` and `WaitReadBuffers()` give us two main advantages:

1. Multiple consecutive blocks can be read with one system call.
2. Advice (hints of future reads) can optionally be issued to the kernel ahead of time.

The traditional `ReadBuffer()` function is now implemented in terms of those functions, to avoid duplication.

A new GUC `io_combine_limit` is defined, and the functions for limiting per-backend pin counts are made into public APIs. Those are provided for use by callers of `StartReadBuffers()`, when deciding how many buffers to read at once. The following commit will add a higher level mechanism for doing that automatically with a practical interface.

With some more infrastructure in later work, `StartReadBuffers()` could be extended to start real asynchronous I/O instead of just issuing advice and leaving `WaitReadBuffers()` to do the work synchronously.

Author: Thomas Munro <thomas.munro@gmail.com>
Author: Andres Freund <andres@anarazel.de> (some optimization tweaks)
Reviewed-by: Melanie Plageman <melanieplageman@gmail.com>
Reviewed-by: Heikki Linnakangas <hlinnaka@iki.fi>
Reviewed-by: Nazir Bilal Yavuz <byavuz81@gmail.com>
Reviewed-by: Dilip Kumar <dilipbalaut@gmail.com>
Reviewed-by: Andres Freund <andres@anarazel.de>
Tested-by: Tomas Vondra <tomas.vondra@enterprisedb.com>
Discussion: <https://postgr.es/m/CA+hUKGJK0i0Ca+mag4BF+2Ho7qo=09CFheB8=g6uT5Um2gkvA@mail.gmail.com>

Векторизованное чтение
нескольких смежных блоков за
ОДИН ВЫЗОВ

Разделение API на два вызова
`StartReadBuffers()` и
`WaitReadBuffers()`: база для
асинхронного IO

Буферный кэш. Загрузка страницы в буфер

```
/*
 * ...
 * StartBufferIO: begin I/O on this buffer
 * In some scenarios there are race conditions in which multiple backends
 * could attempt the same I/O operation concurrently. If someone else
 * has already started I/O on this buffer then we will block on the
 * I/O condition variable until he's done.
 * ...
 */
static bool
StartBufferIO(BufferDesc *buf, bool forInput, bool nowait)
{
    ...
    buf_state = LockBufHdr(buf);
    ...
    buf_state |= BM_IO_IN_PROGRESS;
    UnlockBufHdr(buf, buf_state);
    ResourceOwnerRememberBufferIO(CurrentResourceOwner,
                                   BufferDescriptorGetBuffer(buf));
    ...
}
```

src/backend/storage/buffer/bufmgr.c

Ожидание других процессов строится
на базе условной переменной
(*condition variable*) и под именем
BufferIO (группа IPC)

Перед чтением проставляется флаг
IO_IN_PROGRESS в дескриптор
буфера, с помощью которого все
другие процессы будут ожидать
окончания загрузки

```
/*
 * WaitIO -
 * Block until the IO_IN_PROGRESS flag on 'buf' is cleared.
 */
static void
WaitIO(BufferDesc *buf)
{
    ConditionVariable *cv = BufferDescriptorGetIOCV(buf);

    ConditionVariablePrepareToSleep(cv);
    for (;;)
    {
        uint32 buf_state = LockBufHdr(buf);
        UnlockBufHdr(buf, buf_state);

        if (!(buf_state & BM_IO_IN_PROGRESS))
            break;
        ConditionVariableSleep(cv, WAIT_EVENT_BUFFER_IO);
    }
    ConditionVariableCancelSleep();
}
```

src/backend/storage/buffer/bufmgr.c

Буферный кэш. Clock sweep в PostgreSQL

- Классический Clock-алгоритм предполагает “*second chance*” бит в качестве reference счётчика
- Clock sweep в PostgreSQL счётчик от 0 до 5
 - величина определяет степень привязки страницы к буферу
 - **generalized clock** алгоритм в академической литературе
 - комбинация техник LRU (last recently used) и LFU (least frequently used)
- Терминология в PostgreSQL
 - pin счётчик → *refcount*
 - reference счётчик → *usage count*

Оптимизации кэша. Секционирование BufTable мьютекса (BufferMapping LWLock`a)

src/backend/storage/buffer/README

Notes About Shared Buffer Access Rules

=====

...

* There is a system-wide LWLock, the BufMappingLock, that notionally protects the mapping from buffer tags (page identifiers) to buffers. (Physically, it can be thought of as protecting the hash table maintained by buf_table.c.) To look up whether a buffer exists for a tag, it is sufficient to obtain share lock on the BufMappingLock. Note that one must pin the found buffer, if any, before releasing the BufMappingLock. To alter the page assignment of any buffer, one must hold exclusive lock on the BufMappingLock. This lock must be held across adjusting the buffer's header fields and changing the buf_table hash table. The only common operation that needs exclusive lock is reading in a page that was not in shared buffers already, which will require at least a kernel call and usually a wait for I/O, so it will be slow anyway.

* As of PG 8.2, the BufMappingLock has been split into NUM_BUFFER_PARTITIONS separate locks, each guarding a portion of the buffer tag space. This allows further reduction of contention in the normal code paths. The partition that a particular buffer tag belongs to is determined from the low-order bits of the tag's hash value. The rules stated above apply to each partition independently. If it is necessary to lock more than one partition at a time, they must be locked in partition-number order to avoid risk of deadlock.

BufTableMapping мьютекс
секционирован на
NUM_BUFFER_PARTITIONS
КОЛ-ВО ЛОКОВ

Оптимизации кэша. Секционирование BufTable мьютекса (BufferMapping LWLock`a)

```
/* create a tag so we can lookup the buffer */  
InitBufferTag(&newTag, &smgr->smgr_rlocator.locator,  
forkNum, blockNum);
```

```
/* determine its hash code and partition lock ID */  
newHash = BufTableHashCode(&newTag);  
newPartitionLock = BufMappingPartitionLock(newHash);
```

```
/* see if the block is in the buffer pool already */  
LWLockAcquire(newPartitionLock, LW_SHARED);
```

```
/* Number of partitions of the shared buffer mapping hashtable */  
#define NUM_BUFFER_PARTITIONS 128
```

По хэшу от **BufferTag** (pageid) вычисляется
нужная секция лока:

```
static inline uint32  
BufTableHashPartition(uint32 hashcode)  
{  
    return hashcode % NUM_BUFFER_PARTITIONS;  
}
```

Увеличение кол-ва секций
NUM_BUFFER_PARTITIONS (с
перекомпиляцией исходников) - одна
из оптимизаций **BufferMapping**
ожиданий [*]

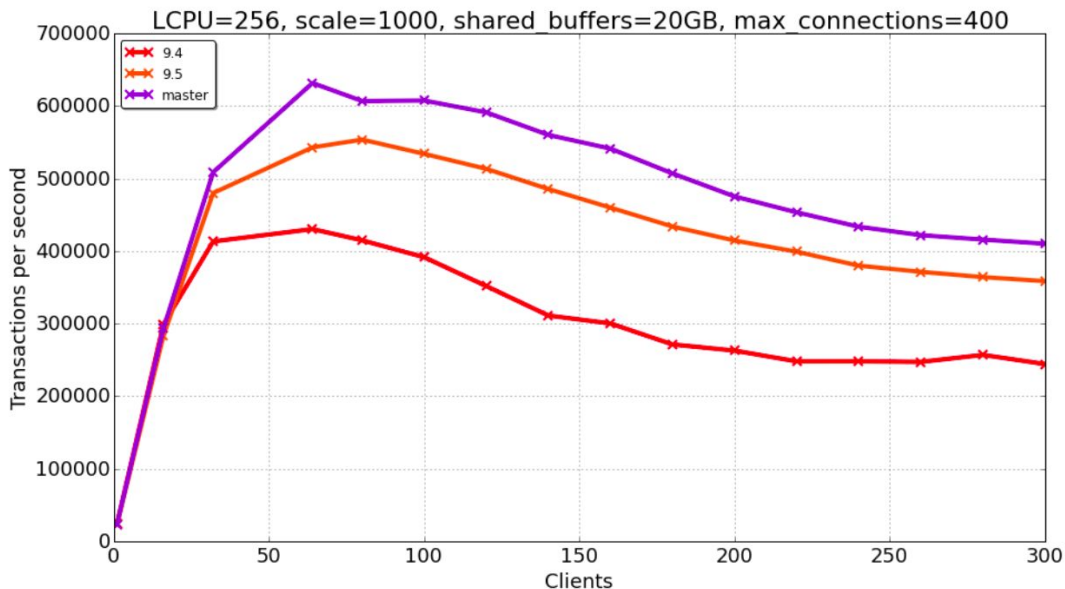
Оптимизации кэша. Lock-free pin/unpin буфера

32.10%	postgres	[.] s_lock
7.77%	postgres	[.] GetSnapshotData
2.64%	postgres	[.] AllocSetAlloc
1.40%	postgres	[.] hash_search_with_hash_value
1.37%	postgres	[.] base_yyparse
1.36%	postgres	[.] SearchCatCache
1.32%	postgres	[.] PinBuffer
1.23%	postgres	[.] LWLockAcquire
1.05%	postgres	[.] palloc
1.01%	postgres	[.] ReadBuffer_common
0.99%	postgres	[.] LWLockRelease

backtrace

```
#0 0x00003fffac40a858 in __newselect_nocancel () from /lib64/po
#1 0x00000000106105f0 in pg_usleep (microsec=<optimized out>)
#2 0x00000000103e5f18 in s_lock (lock=0x3fe607980be0, file=0x1
"bufmgr.c",
line=<optimized out>) at s_lock.c:110
#3 0x00000000103aea10 in UnpinBuffer (buf=0x3fe607980bc0, fix
bufmgr.c:1540
#4 0x00000000103b4910 in ReleaseAndReadBuffer (buffer=<optim
relation=0x3fe6067073e0, blockNum=<optimized out>) at bufmgr.c:
```

- Power8 CPU
 - 2×4×4 ядер
 - smt=8, 2×4×4×8=256 потоков
- workload: pg_bench read-only B-tree search в памяти
- рост до 64 клиентов до 600 тыс. TPS




<https://habr.com/ru/companies/postgrespro/articles/270827/>

<https://shorturl.at/NteSe>

Оптимизации кэша. Lock-free pin буфера

spin ожидание на
снятие бита лока



```
old_buf_state = pg_atomic_read_u32(&buf->state);  
for (;;)   
{  
    if (old_buf_state & BM_LOCKED)  
        old_buf_state = WaitBufHdrUnlocked(buf);  
  
    /* increase refcount and usagecount unless already max. */  
    buf_state = old_buf_state;  
    buf_state += BUF_REFCOUNT_ONE;  
  
    if (BUF_STATE_GET_USAGECOUNT(buf_state) < BM_MAX_USAGE_COUNT)  
        buf_state += BUF_USAGECOUNT_ONE;  
    ...  
    if (pg_atomic_compare_exchange_u32(&buf->state, &old_buf_state,  
                                        buf_state))  
        break;  
}
```

src/backend/storage/buffer/bufmgr.c

```
LockBufHdr(buf);
```

```
buf->refcount++;
```

```
if (buf->usage_count < BM_MAX_USAGE_COUNT)
```

```
    buf->usage_count++;
```

```
UnlockBufHdr(buf);
```



аналогичным образом устроен *unpin*

Оптимизации кэша. Структура атомарной переменной буфер дескриптора

```
typedef struct BufferDesc
```

```
{
```

```
...
```

```
/* state of the tag, containing flags, refcount and usagecount */
```

```
pg_atomic_uint32 state;
```

```
...
```

```
} BufferDesc;
```

```
#define BUF_REFCOUNT_MASK ((1U << 18) - 1)
```

18 бит pin счётчик (refcounter)

```
#define BUF_USAGECOUNT_MASK 0x003C0000U
```

3 бита reference счётчик (usage counter)

```
#define BUF_FLAG_MASK 0xFFC00000U
```

9 бит на флаги, включая бит лока

```
#define BM_LOCKED (1U << 22) /* buffer header is locked */
```

```
#define BM_DIRTY (1U << 23) /* data needs writing */
```

```
#define BM_VALID (1U << 24) /* data is valid */
```

```
#define BM_IO_IN_PROGRESS (1U << 26) /* read or write in progress */
```

```
...
```

Оптимизации кэша. Lock buffer descriptor

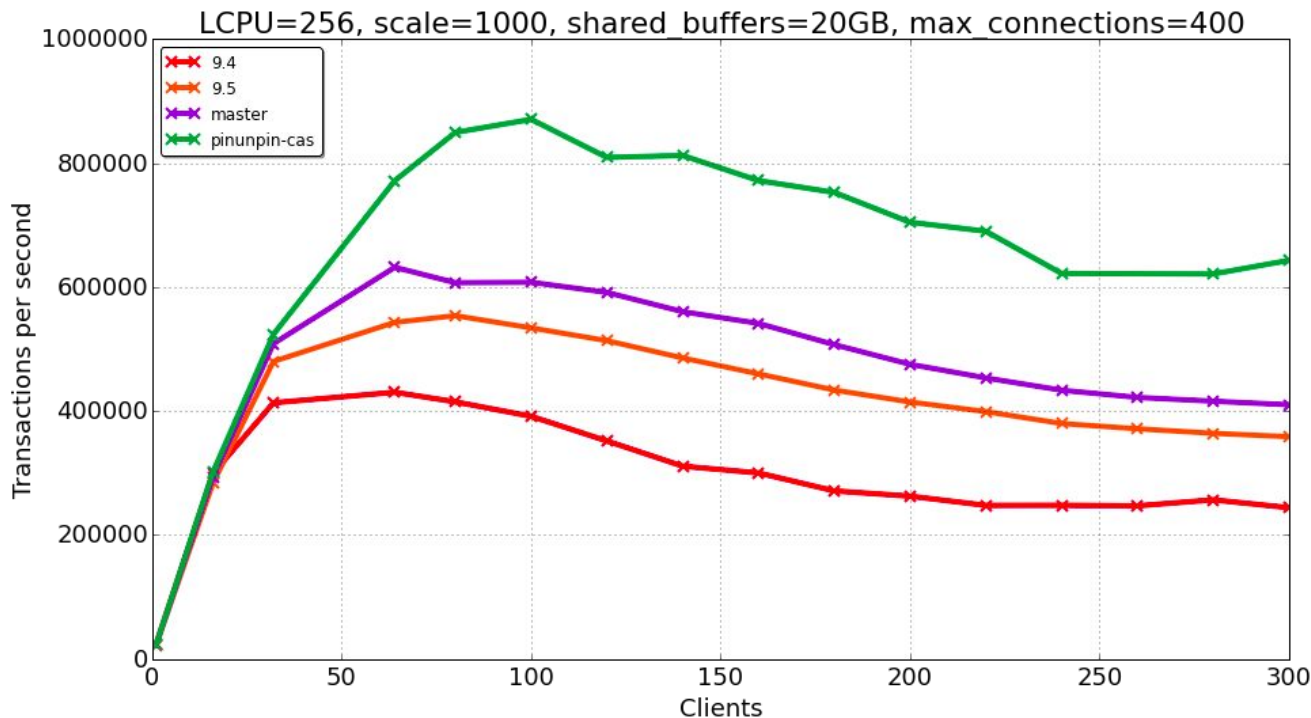
```
/*
 * Lock buffer header - set BM_LOCKED in buffer state.
 */
uint32
LockBufHdr(BufferDesc *desc)
{
    ...
    init_local_spin_delay(&delayStatus);
    while (true)
    {
        /* set BM_LOCKED flag */
        old_buf_state = pg_atomic_fetch_or_u32(&desc->state,
                                                BM_LOCKED);

        /* if it wasn't set before we're OK */
        if (!(old_buf_state & BM_LOCKED))
            break;
        perform_spin_delay(&delayStatus);
    }
    finish_spin_delay(&delayStatus);
    return old_buf_state | BM_LOCKED;
}
```

```
static inline void
UnlockBufHdr(BufferDesc *desc, uint32 buf_state)
{
    pg_write_barrier();
    pg_atomic_write_u32(&desc->state,
                        buf_state & (~BM_LOCKED));
}
```

интерфейс работы со спинлоком дескриптора **сохраняется**

Оптимизации кэша. Lock-free pin/unpin буфера



13.75% postgres [.] GetSnapshotData

4.88% postgres [.] AllocSetAlloc

2.47% postgres [.] LWLockAcquire

рост до 100 клиентов и свыше 800 тыс. TPS

Буферный кэш. Политика вытеснения “ring buffer”

- Решение проблемы “sequential flooding”
 - вымывание кэша при массовом одноразовом чтении/записи (*Sequential Scan* большой таблицы, vacuum, bulk insert)
- Размер таблицы при SeqScan`е должен превосходить **четверть** размера буфера
- Выделяется ограниченный пул буферов на **256Кб**, внутри которых по кольцу происходит вытеснение
- Алгоритм вытеснения встраивается в Clock sweep
 - поддержка *usagecount* счётчика равным единице: расчёт что обход по кольцу случается **до** обхода Clock указателя

Мониторинг и инструментация

- кумулятивные счётчики кэш промахов/попаданий в ***pg_stat**** представлениях
 - на уровне базы *pg_stat_database.blks_read/blks_hit*
 - на уровне таблиц по типам хранилищ *pg_stat_io_table.[heap|idx|toast|tidx]_blks_read/blks_hit*
 - по типу процесса, таблицам, контексту исполнения (normal, vacuum, bulkread, etc.)
pg_stat_io.hits/evictions с PG16
 - счётчик записи буферов на уровне базы *pg_stat_bgwriter.buffer_backend*
- ***pg_stat_statements***: счётчики инструментации по группам запросов (*queryid*)
 - *pg_stat_statements.shared_blks_[hit|read|dirty|written]*
- ***EXPLAIN (ANALYZE, BUFFERS)***: инструментация по конкретному запросу

Инструментация. Tracerepoints

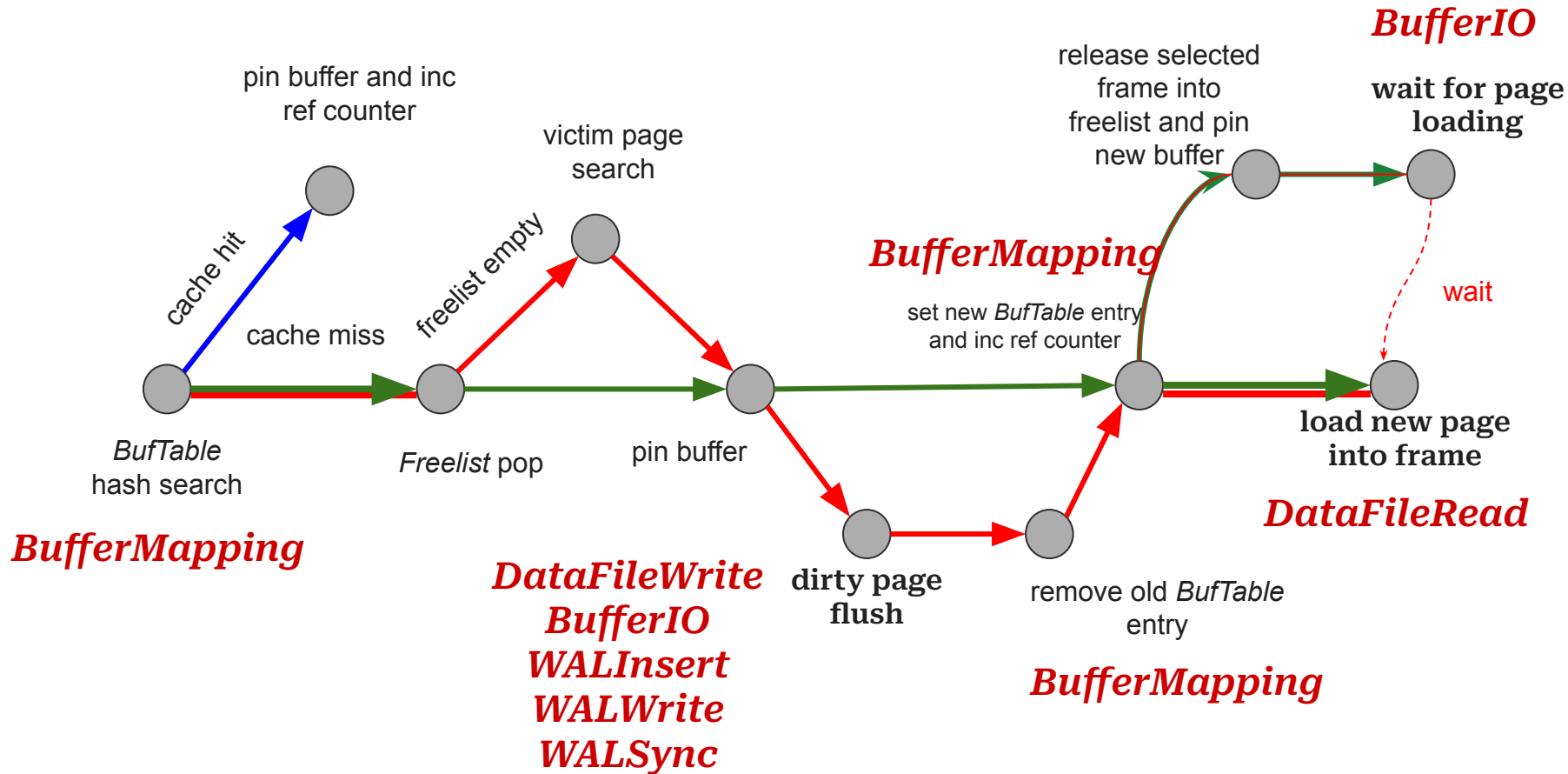
- На аллокацию буфера до загрузки страницы

- `buffer-read-start(ForkNumber, BlockNumber, Oid tablespace, Oid database, Oid relation, int backendID)`
- `buffer-read-done(ForkNumber, BlockNumber, Oid tablespace, Oid database, Oid relation, int backendID)`

- На сброс грязного буфера

- `buffer-flush-start(ForkNumber, BlockNumber, Oid tablespace, Oid database, Oid relation)`
- `buffer-flush-done(ForkNumber, BlockNumber, Oid tablespace, Oid database, Oid relation)`

Инструментация. Ожидания



Полезные инструменты. Расширения

- **pg_buffercache**

- **pg_buffercache** view: просмотр состояния ячеек буфера (содержимое буфер дескрипторов)
- тяжеловерсно для перманентного периодического мониторинга
 - обходит буфер дескрипторы с взятием спинлоков
- функция **pg_buffercache_evict()** для сброса ячейки
 - для записанной страницы отрицательный результат

- **pg_prewarm**

- прогрев буфера, включая автоматический (**autoprewarm**)
- 3 режима:
 - *prefetch* - асинхронный в pagecache ОС (*posix_fadvise(..., POSIX_FADV_WILLNEED)*)
 - *read* - синхронный в pagecache ОС (простым чтением страниц без аллокации в буфер)
 - *buffer* - с сохранением в буфер

Scan sharing

- Переиспользование буфер аллокаций по одинаковым данным от уже запущенных запросов
 - несколько запросов как бы переиспользуют один курсор сканирования по таблице
 - другие названия: *cursor sharing*, *synchronized scan*
- В PostgreSQL
 - работает для больших (более четверти размера буфера) Sequential Scan`ов (bulk read)
 - параметр ***synchronize_seqscans***
- Каждый скан “*leader*” обновляет свою позицию сканирования, привязанную к таблице, в разделяемом LRU кэше
 - сканы “*follower`ы*” начинают table scan с последней позиции по таблице в этом кэше

Спасибо за внимание!
Вопросы. Критика. Пожелания?

milyutinma@gmail.com